pySMT Documentation

Release 0.9.5

Andrea Micheli and Marco Gario

May 05, 2023

Contents

CHAPTER 1

Introduction

pySMT makes working with Satisfiability Modulo Theory simple. pySMT provides an intermediate step between the SMT-LIB (that is universal but not programmable) and solvers API (that are programmable, but specific).

Among others, you can:

- Define formulae in a solver independent way

- Run the same code using multiple solvers

- Easily perform quantifier elimination, interpolation and unsat-core extraction

- Write ad-hoc simplifiers and operators

- and more...

Please let us know of any problem or possible improvements by opening an issue on github or by writing to pysmt@googlegroups.com .

Where to go from here:

- Getting Started: *Installation* and *Hello World*;

- *Tutorials*: Simple examples showing how to perform common operations using pySMT.

- Full *API Reference*

- *Extending pySMT*

Table of Contents

2.1 Getting Started

In this section we will see how to install pySMT and how to solve a simple problem using it.

2.1.1 Installation

To run pySMT you need Python 3.5+ installed. If no solver is installed, pySMT can still create and dump the SMT problem in SMT-LIB format. pySMT works with any SMT-LIB compatible solver. Moreover, pySMT can leverage the API of the following solvers:

- MathSAT (http://mathsat.fbk.eu/)

- Z3 (https://github.com/Z3Prover/z3/)

- CVC4 (http://cvc4.cs.nyu.edu/web/)

- Yices 2 (http://yices.csl.sri.com/)

- CUDD (http://vlsi.colorado.edu/~fabio/CUDD/)

- PicoSAT (http://fmv.jku.at/picosat/)

- Boolector (http://fmv.jku.at/boolector/)

The Python binding for the SMT Solvers must be installed and accessible from your `PYTHONPATH`.

To check which solvers are visible to pySMT, you can use the command `pysmt-install` (simply `install.py` in the sources):

```
$ pysmt-install --check
```

provides the list of installed solvers (and version). Solvers can be installed with the same script: e.g.,

```
$ pysmt-install --msat
```

installs the MathSAT5 solver.

By default the solvers are downloaded, unpacked and built in your home directory in the `.smt_solvers` folder. Compiled libraries and actual solver packages are installed in the relevant `site-packages` directory (e.g. virtual environment's packages root or local user-site).

If you specified a custom `--bindings-path` during install, you can use the `--env` option to obtain a string to update your `PYTHONPATH`. When defaults are used, there's not need to update the `PYTHONPATH`. `pysmt-install` has many options to customize its behavior.

GMPY2

PySMT supports the use of the gmpy2 library (version 2.0.8 or later) to handle multi-precision numbers. This provides an efficient way to perform operations on big numbers, especially when fractions are involved. The gmpy library is used by default if it is installed, otherwise the `fractions` module from Python's standard library is used. The use of the gmpy library can be controlled by setting the system environment variables `PYSMT_GMPY2` to `True` or `False`. If this is set to true and the gmpy library cannot be imported, an exception will be thrown.

Cython

PySMT supports the use of Cython in order to improve the performances of some internal component. If Cython is installed, this will be used (by default) to compile the SMT-LIB parser. The use of Cython can be controlled by setting the environment variable `PYSMT_CYTHON` to `True` or `False`. If set to false, the default pure python implementation will be used.

2.1.2 Hello World

Any decent tutorial starts with a *Hello World* example. We will encode a problem as an SMT problem, and then invoke a solver to solve it. After digesting this example, you will be able to perform the most common operations using pySMT.

The Problem

The problem that we are going to solve is the following:

> Lets consider the letters composing the words *HELLO* and *WORLD*, with a possible integer value between 1 and 10 to each of them.
>
> Is there a value for each letter so that H+E+L+L+O = W+O+R+L+D = 25?

The Basics

The module `pysmt.shortcuts` provides the most used functions of the library. These are simple wrappers around functionalities provided by other objects, therefore, this represents a good starting point if you are interested in learning more about pySMT.

We include the methods to create a new `Symbol()` (i.e., variables), and typing information (the domain of the variable), that is defined in *pysmt.typing*, and we can write the following:

```python
from pysmt.shortcuts import Symbol
from pysmt.typing import INT
```

```
h = Symbol("H", INT)

domain = (1 <= h) & (10 >= h)
```

When importing `pysmt.shortcuts`, the infix notation is enabled by default. Infix notation makes it very easy to experiment with pySMT expressions (e.g., on the shell), however, it tends to make complex code less clear, since it blurs the line between Python operators and SMT operators. In the rest of the example, we will use the textual operators by importing them from `pysmt.shortcuts`.

```
from pysmt.shortcuts import Symbol, LE, GE, And, Int
from pysmt.typing import INT

h = Symbol("H", INT)

# domain = (1 <= h) & (10 >= h)
domain = And(LE(Int(1), h), GE(Int(10), h))
```

Instead of defining one variable at the time, we will use Python's comprehension to apply the same operation to multiple symbols. Comprehensions are so common in pySMT that n-ary operators (such as `And()`, `Or()`, `Plus()`) can accept am iterable object (e.g. lists or generator).

```
from pysmt.shortcuts import Symbol, LE, GE, Int, And, Equals, Plus, Solver
from pysmt.typing import INT

hello = [Symbol(s, INT) for s in "hello"]
world = [Symbol(s, INT) for s in "world"]

letters = set(hello+world)

domains = And(And(LE(Int(1), l),
                  GE(Int(10), l)) for l in letters)

sum_hello = Plus(hello)
sum_world = Plus(world)

problem = And(Equals(sum_hello, sum_world),
              Equals(sum_hello, Int(25)))

formula = And(domains, problem)

print("Serialization of the formula:")
print(formula)
```

Note: Limited serialization

By default printing of a string is limited in depth. For big formulas, you will see something like (`x & (y | ...`
`))`, where deep levels of nestings are replaced with the ellipses `...`. This generally provides you with an idea of how the structure of the formula looks like, without flooding the output when formulas are huge. If you want to print the whole formula, you need to call the `serialize()` method:

```
print(formula) # (x & (y | ... ))
print(formula.serialize()) # (x & (y | (z | y)))
```

Defaults methods for formula allow for simple printing. Checking satisfiability can also be done with a one-liner.

```
print("Checking Satisfiability:")
print(is_sat(formula))
```

Model extraction is provided by the get_model() shortcut: if the formula is unsatisfiable, it will return None, otherwise a *Model*, that is a dict-like structure mapping each Symbol to its value.

```
print("Serialization of the formula:")
print(formula)

print("Checking Satisfiability:")
print(is_sat(formula))
```

Shortcuts are very useful for one off operations. In many cases, however, you want to create an instance of a *Solver* and operate on it incrementally. This can be done using the pysmt.shortcuts.Solver() factory. This factory can be used within a context (with statement) to automatically handle destruction of the solver and associated resources. After creating the solver, we can assert parts of the formula and check their satisfiability. In the following snippet, we first check that the domain formula is satisfiable, and if so, we continue to solve the problem.

```
with Solver(name="z3") as solver:
    solver.add_assertion(domains)
    if not solver.solve():
        print("Domain is not SAT!!!")
        exit()
    solver.add_assertion(problem)
    if solver.solve():
        for l in letters:
            print("%s = %s" %(l, solver.get_value(l)))
    else:
        print("No solution found")
```

In the example, we access the value of each symbol (*get_value()*), however, we can also obtain a model object using *get_model()*.

Note: Incrementality and Model Construction

Many solvers can perform aggressive simplifications if incrementality or model construction are not required. Therefore, if you do not need incrementality and model construction, it is better to call is_sat(), rather than instantiating a solver. Similarly, if you need only one model, you should use get_model()

With pySMT it is possible to run the same code by using different solvers. In our example, we can specify which solver we want to run by changing the way we instantiate it. If any other solver is installed, you can try it by simply changing name="z3" to its codename (e.g., msat):

Solver	pySMT name
MathSAT	msat
Z3	z3
CVC4	cvc4
Yices	yices
Boolector	btor
Picosat	picosat
CUDD	bdd

You can also not specify the solver, and simply state which Logic must be supported by the solver, this will look into the installed solvers and pick one that supports the logic. This might raise an exception (`NoSolverAvailableError`), if no logic for the logic is available.

Here is the complete example for reference using the logic QF_LIA:

```python
from pysmt.shortcuts import Symbol, LE, GE, Int, And, Equals, Plus, Solver
from pysmt.typing import INT

hello = [Symbol(s, INT) for s in "hello"]
world = [Symbol(s, INT) for s in "world"]

letters = set(hello+world)

domains = And(And(LE(Int(1), l),
                  GE(Int(10), l)) for l in letters)

sum_hello = Plus(hello)
sum_world = Plus(world)

problem = And(Equals(sum_hello, sum_world),
              Equals(sum_hello, Int(25)))

formula = And(domains, problem)

print("Serialization of the formula:")
print(formula)

with Solver(logic="QF_LIA") as solver:
    solver.add_assertion(domains)
    if not solver.solve():
        print("Domain is not SAT!!!")
        exit()
    solver.add_assertion(problem)
    if solver.solve():
        for l in letters:
            print("%s = %s" %(l, solver.get_value(l)))
    else:
        print("No solution found")
```

2.1.3 What's Next?

This simple example provides the basic ideas of how to work with pySMT. The best place to understand more about pySMT is the `pysmt.shortcuts` module. All the important functionalities are exported there with a simple to use interface.

To understand more about other functionalities of pySMT, you can take a look at the examples/ folder .

2.2 Tutorials

This page is under-construction. For now, it contains a copy of the files within the examples/ folder of the pySMT repository and a basic tutorial on Boolean logic.

If you are interested in helping us create better tutorials, please let us know at info@pysmt.org .

2.2.1 Boolean Logic in PySMT

An introductory basic example on how to use pySMT for Booean satisfiability: pysmt-tutorials-booleanlogic.

2.2.2 First example

```python
# Checking satisfiability of a formula.
#
# This example shows:
#  1. How to build a formula
#  2. How to perform substitution
#  3. Printing
#  4. Satisfiability checking
from pysmt.shortcuts import Symbol, And, Not, is_sat

varA = Symbol("A") # Default type is Boolean
varB = Symbol("B")
f = And([varA, Not(varB)])
g = f.substitute({varB:varA})

res = is_sat(f)
assert res # SAT
print("f := %s is SAT? %s" % (f, res))

res = is_sat(g)
print("g := %s is SAT? %s" % (g, res))
assert not res # UNSAT
```

2.2.3 Hello World word puzzle

```python
# This is the tutorial example of pySMT.
#
# This example shows how to:
# 1. Deal with Theory atoms
# 2. Specify a solver in the shortcuts (get_model, is_sat etc.)
# 3. Obtain an print a model
#
#
# The goal of the puzzle is to assign a value from 1 to 10 to each letter s.t.
#    H+E+L+L+O = W+O+R+L+D = 25
#
from pysmt.shortcuts import Symbol, And, GE, LT, Plus, Equals, Int, get_model
from pysmt.typing import INT

hello = [Symbol(s, INT) for s in "hello"]
world = [Symbol(s, INT) for s in "world"]
letters = set(hello+world)
domains = And([And(GE(l, Int(1)),
                   LT(l, Int(10))) for l in letters])

sum_hello = Plus(hello) # n-ary operators can take lists
sum_world = Plus(world) # as arguments
problem = And(Equals(sum_hello, sum_world),
              Equals(sum_hello, Int(25)))
formula = And(domains, problem)

print("Serialization of the formula:")
print(formula)

model = get_model(formula)
if model:
    print(model)
else:
    print("No solution found")
```

2.2.4 Hello World word puzzle using infix-notation

```python
# This is a different take on the puzzle.py example
#
# This examples shows how to:
# 1. Enable and use infix notation
# 2. Use a solver context
#
from pysmt.shortcuts import Symbol, And, Plus, Int
from pysmt.shortcuts import Solver
from pysmt.typing import INT

# Infix-Notation is automatically enabled whenever you import pysmt.shortcuts.
#
# To enable it without using shortcuts, do:
#
#   from pysmt.environment import get_env
#   get_env().enable_infix_notation = True
```

```python
#
# Similarly, you can disable infix_notation to prevent its accidental use.
#

hello = [Symbol(s, INT) for s in "hello"]
world = [Symbol(s, INT) for s in "world"]
letters = set(hello+world)

# Infix notation for Theory atoms does the overloading of python
# operator. For boolean connectors, we use e.g., x.And(y) This
# increases readability without running into problems of operator
# precedence.
#
# Note how you can mix prefix and infix notation.
domains = And([(Int(1) <= l).And(Int(10) >= l) for l in letters])

sum_hello = Plus(hello) # n-ary operators can take lists
sum_world = Plus(world) # as arguments
problem = (sum_hello.Equals(sum_world)).And(sum_hello.Equals(Int(25)))
formula = domains.And(problem)

print("Serialization of the formula:")
print(formula)

# A context (with-statment) lets python take care of creating and
# destroying the solver.
with Solver() as solver:
    solver.add_assertion(formula)
    if solver.solve():
        for l in letters:
            print("%s = %s" %(l, solver.get_value(l)))
    else:
        print("No solution found")
```

2.2.5 Combine multiple solvers

```python
# This example requires Z3 and MathSAT to be installed (but you can
#  replace MathSAT with any other solver for QF_LRA)
#
# This examples shows how to:
# 1. Define Real valued constants using floats and fractions
# 2. Perform quantifier elimination
# 3. Pass results from one solver to another
#
from pysmt.shortcuts import Symbol, Or, ForAll, GE, LT, Real, Plus
from pysmt.shortcuts import qelim, is_sat
from pysmt.typing import REAL

x, y, z = [Symbol(s, REAL) for s in "xyz"]

f = ForAll([x], Or(LT(x, Real(5.0)),
                   GE(Plus(x, y, z), Real((17,2))))) # (17,2) ~> 17/2
print("f := %s" % f)
#f := (forall x . ((x < 5.0) | (17/2 <= (x + y + z))))
```

```python
qf_f = qelim(f, solver_name="z3")
print("Quantifier-Free equivalent: %s" % qf_f)
#Quantifier-Free equivalent: (7/2 <= (z + y))

res = is_sat(qf_f, solver_name="msat")
print("SAT check using MathSAT: %s" % res)
#SAT check using MathSAT: True
```

2.2.6 Model-Checking an infinite state system (BMC+K-Induction) in ~150 lines

```python
# This example shows a more advance use of pySMT.
#
# It provides a simple implementation of Bounded Model Checking [1]
# with K-Induction [2], and PDR [3,4], and applies it on a simple
# transition system.
#
# [1] Biere, Cimatti, Clarke, Zhu,
#     "Symbolic Model Checking without BDDs",
#     TACAS 1999
#
# [2] Sheeran, Singh,  Stalmarck,
#     "Checking  safety  properties  using  induction  and  a SAT-solver",
#     FMCAD 2000
#
# [3] Bradley
#     "SAT-Based Model Checking without Unrolling",
#     VMCAI 2011
#
# [4] Een, Mischenko, Brayton
#     "Efficient implementation of property directed reachability",
#     FMCAD 2011
#
from pysmt.shortcuts import Symbol, Not, And, Or, EqualsOrIff, Implies
from pysmt.shortcuts import is_sat, is_unsat, Solver, TRUE
from pysmt.typing import BOOL


def next_var(v):
    """Returns the 'next' of the given variable"""
    return Symbol("next(%s)" % v.symbol_name(), v.symbol_type())


def at_time(v, t):
    """Builds an SMT variable representing v at time t"""
    return Symbol("%s@%d" % (v.symbol_name(), t), v.symbol_type())


class TransitionSystem(object):
    """Trivial representation of a Transition System."""

    def __init__(self, variables, init, trans):
        self.variables = variables
        self.init = init
        self.trans = trans

# EOC TransitionSystem
```

```python
class PDR(object):
    def __init__(self, system):
        self.system = system
        self.frames = [system.init]
        self.solver = Solver()
        self.prime_map = dict([(v, next_var(v)) for v in self.system.variables])

    def check_property(self, prop):
        """Property Directed Reachability approach without optimizations."""
        print("Checking property %s..." % prop)

        while True:
            cube = self.get_bad_state(prop)
            if cube is not None:
                # Blocking phase of a bad state
                if self.recursive_block(cube):
                    print("--> Bug found at step %d" % (len(self.frames)))
                    break
                else:
                    print("   [PDR] Cube blocked '%s'" % str(cube))
            else:
                # Checking if the last two frames are equivalent i.e., are inductive
                if self.inductive():
                    print("--> The system is safe!")
                    break
                else:
                    print("   [PDR] Adding frame %d..." % (len(self.frames)))
                    self.frames.append(TRUE())

    def get_bad_state(self, prop):
        """Extracts a reachable state that intersects the negation
        of the property and the last current frame"""
        return self.solve(And(self.frames[-1], Not(prop)))

    def solve(self, formula):
        """Provides a satisfiable assignment to the state variables that are
→consistent with the input formula"""
        if self.solver.solve([formula]):
            return And([EqualsOrIff(v, self.solver.get_value(v)) for v in self.system.
→variables])
        return None

    def recursive_block(self, cube):
        """Blocks the cube at each frame, if possible.

        Returns True if the cube cannot be blocked.
        """
        for i in range(len(self.frames)-1, 0, -1):
            cubeprime = cube.substitute(dict([(v, next_var(v)) for v in self.system.
→variables]))
            cubepre = self.solve(And(self.frames[i-1], self.system.trans, Not(cube),
→cubeprime))
            if cubepre is None:
                for j in range(1, i+1):
                    self.frames[j] = And(self.frames[j], Not(cube))
                return False
```

```python
            cube = cubepre
        return True

    def inductive(self):
        """Checks if last two frames are equivalent """
        if len(self.frames) > 1 and \
           self.solve(Not(EqualsOrIff(self.frames[-1], self.frames[-2]))) is None:
            return True
        return False

    def __del__(self):
        self.solver.exit()


class BMCInduction(object):

    def __init__(self, system):
        self.system = system

    def get_subs(self, i):
        """Builds a map from x to x@i and from x' to x@(i+1), for all x in system."""
        subs_i = {}
        for v in self.system.variables:
            subs_i[v] = at_time(v, i)
            subs_i[next_var(v)] = at_time(v, i+1)
        return subs_i

    def get_unrolling(self, k):
        """Unrolling of the transition relation from 0 to k:

        E.g. T(0,1) & T(1,2) & ... & T(k-1,k)
        """
        res = []
        for i in range(k+1):
            subs_i = self.get_subs(i)
            res.append(self.system.trans.substitute(subs_i))
        return And(res)

    def get_simple_path(self, k):
        """Simple path constraint for k-induction:
        each time encodes a different state
        """
        res = []
        for i in range(k+1):
            subs_i = self.get_subs(i)
            for j in range(i+1, k+1):
                state = []
                subs_j = self.get_subs(j)
                for v in self.system.variables:
                    v_i = v.substitute(subs_i)
                    v_j = v.substitute(subs_j)
                    state.append(Not(EqualsOrIff(v_i, v_j)))
                res.append(Or(state))
        return And(res)

    def get_k_hypothesis(self, prop, k):
        """Hypothesis for k-induction: each state up to k-1 fulfills the property"""
```

```python
        res = []
        for i in range(k):
            subs_i = self.get_subs(i)
            res.append(prop.substitute(subs_i))
        return And(res)

    def get_bmc(self, prop, k):
        """Returns the BMC encoding at step k"""
        init_0 = self.system.init.substitute(self.get_subs(0))
        prop_k = prop.substitute(self.get_subs(k))
        return And(self.get_unrolling(k), init_0, Not(prop_k))

    def get_k_induction(self, prop, k):
        """Returns the K-Induction encoding at step K"""
        subs_k = self.get_subs(k)
        prop_k = prop.substitute(subs_k)
        return And(self.get_unrolling(k),
                   self.get_k_hypothesis(prop, k),
                   self.get_simple_path(k),
                   Not(prop_k))

    def check_property(self, prop):
        """Interleaves BMC and K-Ind to verify the property."""
        print("Checking property %s..." % prop)
        for b in range(100):
            f = self.get_bmc(prop, b)
            print("   [BMC]    Checking bound %d..." % (b+1))
            if is_sat(f):
                print("--> Bug found at step %d" % (b+1))
                return

            f = self.get_k_induction(prop, b)
            print("   [K-IND]  Checking bound %d..." % (b+1))
            if is_unsat(f):
                print("--> The system is safe!")
                return


def counter(bit_count):
    """Counter example with n bits and reset signal."""

    # Example Counter System (SMV-like syntax)
    #
    # VAR bits: word[bit_count];
    #     reset: boolean;
    #
    # INIT: bits = 0 & reset = FALSE;
    #
    # TRANS: next(bits) = bits + 1
    # TRANS: next(bits = 0) <-> next(reset)

    from pysmt.typing import BVType
    bits = Symbol("bits", BVType(bit_count))
    nbits = next_var(bits)
    reset = Symbol("r", BOOL)
    nreset = next_var(reset)
    variables = [bits, reset]
```

```python
    init = bits.Equals(0) & Not(reset)

    trans = nbits.Equals(bits + 1) &\
            (nbits.Equals(0)).Iff(nreset)

    # A true invariant property: (reset -> bits = 0)
    true_prop = reset.Implies(bits.Equals(0))

    # A false invariant property: (bits != 2**bit_count-1)
    false_prop = bits.NotEquals(2**bit_count -1)

    return (TransitionSystem(variables, init, trans), [true_prop, false_prop])


def main():
    example = counter(4)

    bmcind = BMCInduction(example[0])
    pdr = PDR(example[0])

    for prop in example[1]:
        bmcind.check_property(prop)
        pdr.check_property(prop)
        print("")

if __name__ == "__main__":
    main()
```

2.2.7 How to access functionalities of solvers not currently wrapped by pySMT

```python
# This example requires MathSAT to be installed.
#
# This example shows how to use a solver converter to access
# functionalities of the solver that are not wrapped by pySMT.
#
# Our goal is to call the method msat_all_sat from the MathSAT API.
#
import mathsat
from pysmt.shortcuts import Or, Symbol, Solver, And

def callback(model, converter, result):
    """Callback for msat_all_sat.

    This function is called by the MathSAT API everytime a new model
    is found. If the function returns 1, the search continues,
    otherwise it stops.
    """
    # Elements in model are msat_term .
    # Converter.back() provides the pySMT representation of a solver term.
    py_model = [converter.back(v) for v in model]
    result.append(And(py_model))
    return 1 # go on

x, y = Symbol("x"), Symbol("y")
```

```
f = Or(x, y)

with Solver(name="msat") as msat:
    converter = msat.converter # .converter is a property implemented by all solvers
    msat.add_assertion(f) # This is still at pySMT level

    result = []
    # Directly invoke the mathsat API !!!
    # The second term is a list of "important variables"
    mathsat.msat_all_sat(msat.msat_env(),
        [converter.convert(x)],      # Convert the pySMT term into a MathSAT term
        lambda model : callback(model, converter, result))

    print("'exists y . %s' is equivalent to '%s'" %(f, Or(result)))
    #exists y . (x | y) is equivalent to ((! x) | x)
```

2.2.8 How to use any SMT-LIB compliant SMT solver

```
# This example shows how to define a generic SMT-LIB solver, and use
# it within pySMT. The example looks for mathsat in /tmp, you can
# create a symlink there.
#
# Using this process you can experiment with any SMT-LIB 2 compliant
# solver even if it does not have python bindings, or has not been
# integrated with pySMT.
#
# Note: When using the SMT-LIB wrapper, you can only use logics
# supported by pySMT. If the version of pySMT in use does not support
# Arrays, then you cannot represent arrays.
#

# To define a Generic Solver you need to provide:
#
# - A name to associate to the solver
# - The path to the script + Optional arguments
# - The list of logics supported by the solver
#
# It is usually convenient to wrap the solver in a simple shell script.
# See examples for Z3, Mathsat and Yices in pysmt/test/smtlib/bin/*.template
#
from pysmt.logics import QF_UFLRA, QF_UFIDL, QF_LRA, QF_IDL, QF_LIA
from pysmt.shortcuts import get_env, GT, Solver, Symbol
from pysmt.typing import REAL, INT
from pysmt.exceptions import NoSolverAvailableError

name = "mathsat" # Note: The API version is called 'msat'
path = ["/tmp/mathsat"] # Path to the solver
logics = [QF_UFLRA, QF_UFIDL] # Some of the supported logics

env = get_env()

# Add the solver to the environment
env.factory.add_generic_solver(name, path, logics)

r, s = Symbol("r", REAL), Symbol("s", REAL)
```

```python
p, q = Symbol("p", INT), Symbol("q", INT)

f_lra = GT(r, s)
f_idl = GT(p, q)

# PySMT takes care of recognizing that QF_LRA can be solved by a QF_UFLRA solver.
with Solver(name=name, logic=QF_LRA) as s:
    res = s.solve()
    assert res, "Was expecting '%s' to be SAT" % f_lra


with Solver(name=name, logic=QF_IDL) as s:
    s.add_assertion(f_idl)
    res = s.solve()
    assert res, "Was expecting '%s' to be SAT" % f_idl

try:
    with Solver(name=name, logic=QF_LIA) as s:
        pass
except NoSolverAvailableError:
    # If we ask for a logic that is not contained by any of the
    # supported logics an exception is thrown
    print("%s does not support QF_LIA" % name)
```

2.2.9 How to combine two different solvers to solve an Exists Forall problem

```python
# EF-SMT solver implementation
#
# This example shows:
#  1. How to combine 2 different solvers
#  2. How to extract information from a model
#
from pysmt.shortcuts import Solver, get_model
from pysmt.shortcuts import Symbol, Bool, Real, Implies, And, Not, Equals
from pysmt.shortcuts import GT, LT, LE, Minus, Times
from pysmt.logics import AUTO, QF_LRA
from pysmt.typing import REAL
from pysmt.exceptions import SolverReturnedUnknownResultError


def efsmt(y, phi, logic=AUTO, maxloops=None,
          esolver_name=None, fsolver_name=None,
          verbose=False):
    """Solves exists x. forall y. phi(x, y)"""

    y = set(y)
    x = phi.get_free_variables() - y

    with Solver(logic=logic, name=esolver_name) as esolver:

        esolver.add_assertion(Bool(True))
        loops = 0
        while maxloops is None or loops <= maxloops:
            loops += 1

            eres = esolver.solve()
```

```python
                if not eres:
                    return False
                else:
                    tau = {v: esolver.get_value(v) for v in x}
                    sub_phi = phi.substitute(tau).simplify()
                    if verbose: print("%d: Tau = %s" % (loops, tau))

                    fmodel = get_model(Not(sub_phi),
                                       logic=logic, solver_name=fsolver_name)
                    if fmodel is None:
                        return tau
                    else:
                        sigma = {v: fmodel[v] for v in y}
                        sub_phi = phi.substitute(sigma).simplify()
                        if verbose: print("%d: Sigma = %s" % (loops, sigma))
                        esolver.add_assertion(sub_phi)

        raise SolverReturnedUnknownResultError


def run_test(y, f):
    print("Testing " + str(f))
    try:
        res = efsmt(y, f, logic=QF_LRA, maxloops=20, verbose=True)
        if res == False:
            print("unsat")
        else:
            print("sat : %s" % str(res))
    except SolverReturnedUnknownResultError:
        print("unknown")
    print("\n\n")


def main():
    x,y = [Symbol(n, REAL) for n in "xy"]
    f_sat = Implies(And(GT(y, Real(0)), LT(y, Real(10))),
                    LT(Minus(y, Times(x, Real(2))), Real(7)))

    f_incomplete = And(GT(x, Real(0)), LE(x, Real(10)),
                       Implies(And(GT(y, Real(0)), LE(y, Real(10)),
                                   Not(Equals(x, y))),
                               GT(y, x)))

    run_test([y], f_sat)
    run_test([y], f_incomplete)


if __name__ == "__main__":
    main()
```

2.2.10 How to detect the logic of a formula and perform model enumeration

```python
# Perform ALL-SMT on Theory atoms
#
# This example shows:
```

```python
#   - How to get the logic of a formula
#   - How to enumerate models
#   - How to extract a partial model
#   - How to use the special operator EqualsOrIff
#
from pysmt.shortcuts import Solver, Not, And, Symbol, Or
from pysmt.shortcuts import LE, GE, Int, Plus, Equals, EqualsOrIff
from pysmt.typing import INT
from pysmt.oracles import get_logic


def all_smt(formula, keys):
    target_logic = get_logic(formula)
    print("Target Logic: %s" % target_logic)
    with Solver(logic=target_logic) as solver:
        solver.add_assertion(formula)
        while solver.solve():
            partial_model = [EqualsOrIff(k, solver.get_value(k)) for k in keys]
            print(partial_model)
            solver.add_assertion(Not(And(partial_model)))


A0 = Symbol("A0", INT)
A1 = Symbol("A1", INT)
A2 = Symbol("A2", INT)

f = And(GE(A0, Int(0)), LE(A0, Int(5)),
        GE(A1, Int(0)), LE(A1, Int(5)),
        GE(A2, Int(0)), LE(A2, Int(5)),
        Equals(Plus(A0, A1, A2), Int(8)))

all_smt(f, [A0, A1, A2])

# By using the operator EqualsOrIff, we can mix theory and bool variables
x = Symbol("x")
y = Symbol("y")
f = And(f, Or(x,y))

all_smt(f, [A0, A1, A2, x])
```

2.2.11 Shows how to use multi-processing to perform parallel and asynchronous solving

```python
# This example requires Z3
#
# This example shows how to parallelize solving using pySMT, by using
# the multiprocessing library.
#
# All shortcuts (is_sat, is_unsat, get_model, etc.) can be easily
# used. We sho two techniques: map and apply_async.
#
# Map applies a given function to a list of elements, and returns a
# list of results. We show an example using is_sat.
#
# Apply_async provides a way to perform operations in an asynchronous
```

```python
# way. This allows us to start multiple solving processes in parallel,
# and collect the results whenever they become available.
#
# More information can be found in the Python Standard Library
# documentation for the multiprocessing module.
#
#
# NOTE: When running this example, a warning will appear saying:
#    "Contextualizing formula during is_sat"
#
# Each process will inherit a different formula manager, but the
# formulas that we are solving have been all created in the same
# formula manager. pySMT takes care of moving the formula across
# formula managers. This step is called contextualization, and occurs
# only when using multiprocessing.
# To disable the warning see the python module warnings.
#
from multiprocessing import Pool, TimeoutError
from time import sleep

from pysmt.test.examples import get_example_formulae
from pysmt.shortcuts import is_sat, is_valid, is_unsat
from pysmt.shortcuts import And

# Ignore this for now
def check_validity_and_test(args):
    """Checks expression and compare the outcome against a known value."""

    expr, expected = args # IMPORTANT: Unpack args !!!
    local_res = is_valid(expr)
    return local_res == expected


# Create the Pool with 4 workers.
pool = Pool(4)

# We use the examples formula from the test suite.
# This generator iterates over all the expressions
f_gen = (f.expr for f in get_example_formulae())

# Call the functino is_sat on each expression
res = pool.map(is_sat, f_gen)

# The result is a list of True/False, in the same order as the input.
print(res)

sleep(1) # Have some time to look at the result

# Notice that all shortcuts (is_sat, qelim, etc) require only one
# mandatory argument. To pass multiple arguments, we need to pack them
# together.

# Lets create a list of (formula, result), where result is the
# expected result of the is_valid query.
f_gen = (f.expr for f in get_example_formulae())
res_gen = (f.is_valid for f in get_example_formulae())
args_gen = zip(f_gen, res_gen)
```

```python
# We now define a function that check the formula against the expected
# value of is_valid: See check_validity_and_test(...) above.
# Due to the way multiprocessing works, we need to define the function
# _before_ we create the Pool.

# As before, we call the map on the pool
res = pool.map(check_validity_and_test, args_gen)

# We can also apply a map-reduce strategy, by making sure that all
# results are as expected.
if all(res):
    print("Everything is ok!")
else:
    print("Ooops, something is wrong!")
    print(res)

# A different option is to run solvers in an asynchronous way.  This
# does not provide us with any guarantee on the execution order, but
# it is particular convenient, if we want to perform solving together
# with another long operation (e.g., I/O, network etc.)  or if we want
# to run multiple solvers.

# Create a formula
big_f = And(f.expr for f in get_example_formulae() \
            if not f.logic.theory.bit_vectors and \
                not f.logic.theory.arrays and \
                not f.logic.theory.strings and \
                    f.logic.theory.linear)

# Create keyword arguments for the function call.
# This is the simplest way to pass multiple arguments to apply_async.
kwargs = {"formula": big_f, "solver_name": "z3"}
future_res_sat = pool.apply_async(is_sat, kwds=kwargs)
future_res_unsat = pool.apply_async(is_unsat, kwds=kwargs)

# In the background, the solving is taking place... We can do other
# stuff in the meanwhile.
print("This is non-blocking...")

# Get the result with a deadline.
# See multiprocessing.pool.AsyncResult for more options
sat_res = future_res_sat.get(10)  # Get result after 10 seconds or kill
try:
    unsat_res = future_res_unsat.get(0) # No wait
except TimeoutError:
    print("UNSAT result was not ready!")
    unsat_res = None
print(sat_res, unsat_res)
```

2.2.12 Demonstrates how to perform SMT-LIB parsing, dumping and extension

```python
# The last part of the example requires a QF_LIA solver to be installed.
#
#
```

```python
# This example shows how to interact with files in the SMT-LIB
# format. In particular:
#
# 1. How to read a file in SMT-LIB format
# 2. How to write a file in SMT-LIB format
# 3. Formulas and SMT-LIB script
# 4. How to access annotations from SMT-LIB files
# 5. How to extend the parser with custom commands
#
from io import StringIO

from pysmt.smtlib.parser import SmtLibParser


# To make the example self contained, we store the example SMT-LIB
# script in a string.
DEMO_SMTLIB=\
"""
(set-logic QF_LIA)
(declare-fun p () Int)
(declare-fun q () Int)
(declare-fun x () Bool)
(declare-fun y () Bool)
(define-fun .def_1 () Bool (! (and x y) :cost 1))
(assert (=> x (> p q)))
(check-sat)
(push)
(assert (=> y (> q p)))
(check-sat)
(assert .def_1)
(check-sat)
(pop)
(check-sat)
"""

# We read the SMT-LIB Script by creating a Parser.
# From here we can get the SMT-LIB script.
parser = SmtLibParser()

# The method SmtLibParser.get_script takes a buffer in input. We use
# StringIO to simulate an open file.
# See SmtLibParser.get_script_fname() if to pass the path of a file.
script = parser.get_script(StringIO(DEMO_SMTLIB))

# The SmtLibScript provides an iterable representation of the commands
# that are present in the SMT-LIB file.
#
# Printing a summary of the issued commands
for cmd in script:
    print(cmd.name)
print("*"*50)

# SmtLibScript provides some utilities to perform common operations: e.g,
#
# - Checking if a command is present
assert script.contains_command("check-sat")
# - Counting the occurrences of a command
```

```python
assert script.count_command_occurrences("assert") == 3
# - Obtain all commands of a particular type
decls = script.filter_by_command_name("declare-fun")
for d in decls:
    print(d)
print("*"*50)


# Most SMT-LIB scripts define a single SAT call. In these cases, the
# result can be obtained by conjoining multiple assertions.  The
# method to do that is SmtLibScript.get_strict_formula() that, raises
# an exception if there are push/pop calls.  To obtain the formula at
# the end of the execution of the Script (accounting for push/pop) we
# use get_last_formula
#
f = script.get_last_formula()
print(f)


# Finally, we serialize the script back into SMT-Lib format.  This can
# be dumped into a file (see SmtLibScript.to_file).  The flag daggify,
# specifies whether the printing is done as a DAG or as a tree.

buf_out = StringIO()
script.serialize(buf_out, daggify=True)
print(buf_out.getvalue())


print("*"*50)
# Expressions can be annotated in order to provide additional
# information. The semantic of annotations is solver/problem
# dependent. For example, VMT uses annotations to identify two
# expressions as 1) the Transition Relation and 2) Initial Condition
#
# Here we pretend that we make up a ficticious Weighted SMT format
# and label .def1 with cost 1
#
# The class pysmt.smtlib.annotations.Annotations deals with the
# handling of annotations.
#
ann = script.annotations
print(ann.all_annotated_formulae("cost"))


print("*"*50)

# Annotations are part of the SMT-LIB standard, and are the
# recommended way to perform inter-operable operations. However, in
# many cases, we are interested in prototyping some algorithm/idea and
# need to write the input files by hand. In those cases, using an
# extended version of SMT-LIB usually provides a more readable input.
# We provide now an example on how to define a symbolic transition
# system as an extension of SMT-LIB.
# (A more complete version of this example can be found in :
#    pysmt.tests.smtlib.test_parser_extensibility.py)
#
EXT_SMTLIB="""\
(declare-fun A () Bool)
(declare-fun B () Bool)
(init (and A B))
(trans (=> A (next A)))
```

```
(exit)
"""

# We define two new commands (init, trans) and a new
# operator (next). In order to parse this file, we need to create a
# sub-class of the SmtLibParser, and add handlers for he new commands
# and operators.
from pysmt.smtlib.script import SmtLibCommand


class TSSmtLibParser(SmtLibParser):
    def __init__(self, env=None, interactive=False):
        SmtLibParser.__init__(self, env, interactive)

        # Add new commands
        #
        # The mapping function takes care of consuming the command
        # name from the input stream, e.g., '(init' . Therefore,
        # _cmd_init will receive the rest of the stream, in our
        # example, '(and A B)) ...'
        self.commands["init"] = self._cmd_init
        self.commands["trans"] = self._cmd_trans

        # Remove unused commands
        #
        # If some commands are not compatible with the extension, they
        # can be removed from the parser. If found, they will cause
        # the raising of the exception UnknownSmtLibCommandError
        del self.commands["check-sat"]
        del self.commands["get-value"]
        # ...

        # Add 'next' function
        #
        # New operators can be added similarly as done for commands.
        # e.g., 'next'. The helper function _operator_adapter,
        # simplifies the writing of such extensions.  In this example,
        # we will rewrite the content of the next without the need of
        # introducing a new pySMT operator. If you are interested in a
        # simple way of handling new operators in pySMT see
        # pysmt.test.test_dwf.
        self.interpreted["next"] = self._operator_adapter(self._next_var)

    def _cmd_init(self, current, tokens):
        # This cmd performs the parsing of:
        #   <expr> )
        # and returns a new SmtLibCommand
        expr = self.get_expression(tokens)
        self.consume_closing(tokens, current)
        return SmtLibCommand(name="init", args=(expr,))

    def _cmd_trans(self, current, tokens):
        # This performs the same parsing as _cmd_init, but returns a
        # different object. The parser is not restricted to return
        # SmtLibCommand, but using them makes handling them
        # afterwards easier.
        expr = self.get_expression(tokens)
        self.consume_closing(tokens, current)
```

```python
        return SmtLibCommand(name="trans", args=(expr,))


    def _next_var(self, symbol):
        # The function is called with the arguments obtained from
        # parsing the rest of the SMT-LIB file. In this case, 'next'
        # is a unary function, thus we have only 1 argument. 'symbol'
        # is an FNode. We require that 'symbol' is _really_ a symbol:
        if symbol.is_symbol():
            name = symbol.symbol_name()
            ty = symbol.symbol_type()
            # The return type MUST be an FNode, because this is part
            # of an expression.
            return self.env.formula_manager.Symbol("next_" + name, ty)
        else:
            raise ValueError("'next' operator can be applied only to symbols")


    def get_ts(self, script):
        # New Top-Level command that takes a script stream in input.
        # We return a pair (Init, Trans) that defines the symbolic
        # transition system.
        init = self.env.formula_manager.TRUE()
        trans = self.env.formula_manager.TRUE()

        for cmd in self.get_command_generator(script):
            if cmd.name=="init":
                init = cmd.args[0]
            elif cmd.name=="trans":
                trans = cmd.args[0]
            else:
                # Ignore other commands
                pass

        return (init, trans)


# Time to try out the parser !!!
#
# First we check that the standard SMT-Lib parser cannot handle the new format.
from pysmt.exceptions import UnknownSmtLibCommandError

try:
    parser.get_script(StringIO(EXT_SMTLIB))
except UnknownSmtLibCommandError as ex:
    print("Unsupported command: %s" % ex)

# The new parser can parse our example, and returns the (init, trans) pair
ts_parser = TSSmtLibParser()
init, trans = ts_parser.get_ts(StringIO(EXT_SMTLIB))
print("INIT: %s" % init.serialize())
print("TRANS: %s" % trans.serialize())
```

2.2.13 Shows the use of UNSAT Core as debugging tools

```python
#
# This example requires MathSAT or Z3
#
```

```python
# In this example, we will encode a more complex puzzle and see:
#
# - Use of UNSAT cores as a debugging tool
# - Conjunctive partitioning
# - Symbol handling delegation to auxiliary functions
#

# This puzzle is known as Einstein Puzzle
#
# There are five houses in five different colours in a row. In each
# house lives a person with a different nationality. The five owners
# drink a certain type of beverage, smoke a certain brand of cigar and
# keep a certain pet.
#
# No owners have the same pet, smoke the same brand of cigar, or drink
# the same beverage.
#
# The Brit lives in the red house.
# The Swede keeps dogs as pets.
# The Dane drinks tea.
# The green house is on the immediate left of the white house.
# The green house owner drinks coffee.
# The owner who smokes Pall Mall rears birds.
# The owner of the yellow house smokes Dunhill.
# The owner living in the center house drinks milk.
# The Norwegian lives in the first house.
# The owner who smokes Blends lives next to the one who keeps cats.
# The owner who keeps the horse lives next to the one who smokes Dunhill.
# The owner who smokes Bluemasters drinks beer.
# The German smokes Prince.
# The Norwegian lives next to the blue house.
# The owner who smokes Blends lives next to the one who drinks water.
#
# The question is: who owns the fish?

from pysmt.shortcuts import (Symbol, ExactlyOne, Or, And, FALSE, TRUE,
                             Iff, Implies)
from pysmt.shortcuts import get_model, get_unsat_core, is_sat, is_unsat


#
# Lets start by expliciting all values for all dimensions

Color = "white", "yellow", "blue", "red", "green"
Nat = "german", "swedish", "british", "norwegian", "danish"
Pet =  "birds", "cats", "horses", "fish", "dogs"
Drink = "beer", "water", "tea", "milk", "coffee"
Smoke = "blends", "pall_mall", "prince", "bluemasters", "dunhill"
Houses = range(0,5)
#
# We number the houses from 0 to 4, and create the macros to assert
# properties of the i-th house:
#
#  e.g., color(1, "green")  to indicate that the house 1 is Green
#
# This is not strictly necessary, but it is a way of making programs
# more readable.
#
```

```python
def color(number, name):
    assert name in Color
    if number in Houses:
        return Symbol("%d_color_%s" % (number, name))
    return FALSE()


def nat(number, name):
    assert name in Nat
    if number in Houses:
        return Symbol("%d_nat_%s" % (number, name))
    return FALSE()


def pet(number, name):
    assert name in Pet
    if number in Houses:
        return Symbol("%d_pet_%s" % (number, name))
    return FALSE()


def drink(number, name):
    assert name in Drink
    if number in Houses:
        return Symbol("%d_drink_%s" % (number, name))
    return FALSE()


def smoke(number, name):
    assert name in Smoke
    if number in Houses:
        return Symbol("%d_smoke_%s" % (number, name))
    return FALSE()

#
# We can encode the facts
#

facts = TRUE()
for i in Houses:
    facts = (facts &
        # The Brit lives in the red house.
        nat(i, "british").Iff(color(i, "red")) &

        # The Swede keeps dogs as pets.
        nat(i, "swedish").Iff(pet(i, "dogs")) &

        # The Dane drinks tea.
        nat(i, "danish").Iff(drink(i, "tea")) &

        # The green house is on the immediate left of the white house.
        color(i, "green").Iff(color(i+1, "white")) &

        # The green house owner drinks coffee.
        color(i, "green").Iff(drink(i, "coffee")) &

        # The owner who smokes Pall Mall rears birds.
        smoke(i, "pall_mall").Iff(pet(i, "birds")) &

        # The owner of the yellow house smokes Dunhill.
        color(i, "yellow").Iff(smoke(i, "dunhill")) &
```

```python
        # The owner who smokes Bluemasters drinks beer.
        smoke(i, "bluemasters").Iff(drink(i, "beer")) &

        # The German smokes Prince.
        nat(i, "german").Iff(smoke(i, "prince")) &

        # The owner who smokes Blends lives next to the one who keeps cats.
        smoke(i, "blends").Implies(pet(i-1, "cats") | pet(i+1, "cats")) &

        # The owner who keeps the horse lives next to the one who smokes Dunhill.
        pet(i, "horses").Implies(smoke(i-1, "dunhill") | smoke(i+1, "dunhill")) &

        # The Norwegian lives next to the blue house.
        # Carefull with this one!
        nat(i, "norwegian").Iff(color(i-1, "blue") | color(i+1, "blue")) &

        # The owner who smokes Blends lives next to the one who drinks water.
        smoke(i, "blends").Implies(drink(i-1, "water") | drink(i+1, "water"))
    )

facts = And(facts,
            # The owner living in the center house drinks milk.
            drink(2, "milk"),

            # The Norwegian lives in the first house.
            nat(0, "norwegian")
)


domain = And(
    And(ExactlyOne(color(i, c) for i in Houses) for c in Color),
    And(ExactlyOne(nat(i, c) for i in Houses) for c in Nat),
    And(ExactlyOne(pet(i, c) for i in Houses) for c in Pet),
    And(ExactlyOne(drink(i, c) for i in Houses) for c in Drink),
    And(ExactlyOne(smoke(i, c) for i in Houses) for c in Smoke),
    #
    And(ExactlyOne(color(i, c) for c in Color) for i in Houses),
    And(ExactlyOne(nat(i, c) for c in Nat) for i in Houses),
    And(ExactlyOne(pet(i, c) for c in Pet) for i in Houses),
    And(ExactlyOne(drink(i, c) for c in Drink) for i in Houses),
    And(ExactlyOne(smoke(i, c) for c in Smoke) for i in Houses),
    )

problem = domain.And(facts)

model = get_model(problem)

if model is None:
    print("UNSAT")
    # We first check whether the constraints on the domain and problem
    # are satisfiable in isolation.
    assert is_sat(facts)
    assert is_sat(domain)
    assert is_unsat(problem)

    # In isolation they are both fine, rules from both are probably
```

```python
        # interacting.
        #
        # The problem is given by a nesting of And().
        # conjunctive_partition can be used to obtain a "flat"
        # structure, i.e., a list of conjuncts.
        #
        from pysmt.rewritings import conjunctive_partition
        conj = conjunctive_partition(problem)
        ucore = get_unsat_core(conj)
        print("UNSAT-Core size '%d'" % len(ucore))
        for f in ucore:
            print(f.serialize())


        # The exact version of the UNSAT-Core depends on the solver in
        # use.  Nevertheless, this represents a starting point for your
        # debugging.  A possible way to approach the result is to look for
        # clauses of size 1 (i.e., unit clauses). In the facts list there
        # is only 1 fact:
        #   0_nat_norwegian
        #
        # The clause ("1_color_blue" <-> "0_nat_norwegian")
        # Implies that "1_color_blue"
        # But (("3_color_blue" | "1_color_blue") <-> "2_nat_norwegian")
        # Requires "2_nat_norwegian"
        # The ExactlyOne constraint forbids that both 0 and 2 are nowegian
        # thus, we have a better idea of where the problem might be.
        #
        # Please go back to the comment '# Careful with this!!!' in the
        # facts list, and change the Iff with an Implies.
        #
        # Done?
        #
        # Good, you should be getting a model, now!
else:
    for h in Houses:
        # Extract the relevants bits to get some pretty-printing
        c = [x for x in Color if model[color(h, x)].is_true()][0]
        n = [x for x in Nat if model[nat(h, x)].is_true()][0]
        p = [x for x in Pet if model[pet(h, x)].is_true()][0]
        d = [x for x in Drink if model[drink(h, x)].is_true()][0]
        s = [x for x in Smoke if model[smoke(h, x)].is_true()][0]
        print(h, c, n, p, d, s)
        if p == "fish":
            sol = "The '%s' owns the fish!" % n
    print(sol)
```

2.3 Change Log

2.3.1 0.9.5: 2022-05-28 – 2 years of bugfixes

Intermediate release that collects 2 years of bugfixes and improvements.

Python 2 was deprecated in version 0.9.0, and this version removes the use of compatible code for that version.

What's Changed

- Add support for boolean-typed array in the AtomsOracle by @mikand in https://github.com/pysmt/pysmt/pull/644

- Switched from nosetests to pytest by @mikand in https://github.com/pysmt/pysmt/pull/662

- Fixed a bug in yices quantifier support and added regression test by @mikand in https://github.com/pysmt/pysmt/pull/657

- Fix Boolector install script by @4tXJ7f in https://github.com/pysmt/pysmt/pull/656

- BUG: define UFNIA as logic with integer arithmetic by @johnyf in https://github.com/pysmt/pysmt/pull/659

- Handling of algebraic constants in simplify by @EnricoMagnago in https://github.com/pysmt/pysmt/pull/658

- Integer div by @EnricoMagnago in https://github.com/pysmt/pysmt/pull/667

- Fix CVC4 installation on macOS by @kammoh in https://github.com/pysmt/pysmt/pull/666

- Bug in times distributor by @EnricoMagnago in https://github.com/pysmt/pysmt/pull/671

- Fixed reset_assertion method for incremental-tracking solvers by @mikand in https://github.com/pysmt/pysmt/pull/672

- Minor Corrections by @mfarif in https://github.com/pysmt/pysmt/pull/673

- implement add_assertions method for solver. by @EnricoMagnago in https://github.com/pysmt/pysmt/pull/679

- Fix "get_model" when called from a generic solver (Fix #674) by @btwael in https://github.com/pysmt/pysmt/pull/675

- Remove six and python 2 compatibility code by @marcogario in https://github.com/pysmt/pysmt/pull/684

- Added fallback to Swig3 to address as much as possible issue #682 by @mikand in https://github.com/pysmt/pysmt/pull/685

- Fix to correctly pass logic to solvers started by Portfolio by @ekilmer in https://github.com/pysmt/pysmt/pull/683

- SmtLib model validation support by @mikand in https://github.com/pysmt/pysmt/pull/681

- Fix iss694 by @EnricoMagnago in https://github.com/pysmt/pysmt/pull/695

- Fixed CVC4 installer after upstream repository renaming by @mikand in https://github.com/pysmt/pysmt/pull/697

- Remove call to *FNode.substitute* in SmtLibExecutionCache by @EnricoMagnago in https://github.com/pysmt/pysmt/pull/699

- Added printing of annotations to smt lib printers by @agirardi-fbk in https://github.com/pysmt/pysmt/pull/703

- Integer div by @EnricoMagnago in https://github.com/pysmt/pysmt/pull/705

- Updated docker images to solve deprecation issue on azure pipelines by @mikand in https://github.com/pysmt/pysmt/pull/706

- Workaround to fix Z3 segfault by @mikand in https://github.com/pysmt/pysmt/pull/713

- Add possibility to use several BV operators as left associative by @agirardi-fbk in https://github.com/pysmt/pysmt/pull/714

- Fixed issue #613 by @mikand in https://github.com/pysmt/pysmt/pull/710

New Contributors * @4tXJ7f made their first contribution in https://github.com/pysmt/pysmt/pull/656 * @johnyf made their first contribution in https://github.com/pysmt/pysmt/pull/659 * @kammoh made their first contribution in https://github.com/pysmt/pysmt/pull/666 * @mfarif made their first contribution in https://github.com/pysmt/pysmt/pull/673 * @btwael made their first contribution in https://github.com/pysmt/pysmt/pull/675 * @ekilmer made their

first contribution in https://github.com/pysmt/pysmt/pull/683 * @agirardi-fbk made their first contribution in https://github.com/pysmt/pysmt/pull/703

Full Changelog: https://github.com/pysmt/pysmt/compare/v0.9.0. . . v0.9.5

2.3.2 0.9.0: 2020-04-26 – PySMT as module

General:

- **PySMT as module (PR #573): It is now possible to do** python -m pysmt install

  and

     python -m pysmt shell

  while the tool pystm-install is still available, using the module syntax makes it easier to understand which version of python will be used, in case multiple interpreters co-exist.

- Added functions to obtain the symbols in SMTLIB script (PR #583)

- Python 2 is not supported anymore. While it might still work, we will not actively test for it anymore.

Solvers:

- Boolector: Incremental and Unsat cores support (PR #591, #567). Thanks **Makai Mann** for providing the patch.

- Picosat: Fixed a bug related to solver reset (PR #567)

- Boolector: Upgrade to 3.2.1

- MathSAT: Upgrade to 5.6.1

Bugfix:

- Collections.abc: fix deprecation warning (PR #574, PR #562). Thanks to **Liana Hadarean** and **Caleb Donovick**.

- PysmtSyntaxError: Fix missing message constructor (PR #576). Thanks to **Liana Hadarean** for providing a fix.

- Version: Static version in make_distrib.sh (PR #575)

- Fix simplifier StrIndexOf capitalization (PR #563). Thanks to **Makai Mann** for providing the patch.

- Sort the arguments of Times while simplifying (PR #561). Thanks to **Ahmed Irfan** for providing the patch.

- Fix bug in deque pop in smtlib/parser (PR #558). Thanks to **Sebastiano Mariani** for providing the patch.

- Function names quoted with ' instead of | when seraializing to smt2 (PR #584). Thanks **Kevin Laeufer** for reporting this.

- Fix assertion tracking for boolector (PR #589). Thanks **Makai Mann** for providing the patch.

- Handle one-bit shifts in btor (PR #592) Thanks **Makai Mann** for providing the patch.

- Fix issue with bv conversion in Yices (PR #597). Thanks to @*nano-o* for reporting this.

- Fix Mathsat signature for BV_CONCAT (PR #598). Thanks **Makai Mann** for providing the patch.

- Support n-ary BVConcat (PR #621). Thanks **Ridwan Shariffdeen** for reporting this.

- Fix a correctness issue when reading from SMT-LIB interface (Issue #616, #615, PR #619). Thanks to **Sergio Mover** for reporting this.

- Clear pending assertions in IncrementalTrackingSolver.assertions (PR #627). Thanks to **Enrico Magnago** for reporting this.

- Various documentation fixes. Thanks to **Matthew Fernandez**, **Guillem Francès**, and **Gianluca Redondi**.

- Disable multiprocessing in run_tests.sh script (PR #637). Thanks **Patrick Trentin** for reporting this.

2.3.3 0.8.0: 2019-01-27 – Better Install and Great Community

BACKWARDS INCOMPATIBLE CHANGES:

> Disabled support for interpolation in Z3, since this is not available anymore up-stream

Deprecation Warning:

This release is the **last** release to support Python 2.7. Starting from 0.9.0 only Python 3+ will be supported.

General:

- Solver installation within site-package (PR #517). pysmt-install now installs the solvers within the site-package directory (by default). This makes it possible to work with virtual environments, and does not require anymore to export the Python path, greatly simplifying the installation process. Thanks to **Radomir Stevanovic** for contributing the patch.

- Simplify shared lib usage (PR #494): Modify z3 and msat installers in order to make their shared binary objects (libraries/dlls) auto-discoverable, without the need for setting LD_LIBRARY_PATH/PATH. Thanks to **Radomir Stevanovic** for contributing the patch.

- BV Simplification (PR #531): Multiple improvements on the simplification of BV expressions. Thanks to **Haozhong Zhang** for contributing the patch.

- Ackermannization (PR #515): Add support for Ackermannization in pysmt.rewritings. Thanks to **Yoni Zohar** for contributing the patch.

- FNode.bv_str: Multiple format for BV printing (PR #468)

- Examples (PR #507): Extend model_checking example with PDR. Thanks to **Cristian Mattarei** for contributing the patch.

- Docs: Tutorial on basic boolean solving (PR #535)

- Tests: Removed old warning and other clean-ups (PR #532, #512)

- Warnings (PR #497): Importing pysmt.shortcuts will only raise warnings within pySMT, instead of all warnings from external libraries.

- Examples (PR #541): Add example for the theory of Strings

- Top-Level Propagator (PR #544): Add a basic toplevel-propagation functionality to propagate definitions of the form: variable = constant, variable = variable, constant = constant . Thanks to **Ahmed Irfan** for providing this feature.

- Clean-up debug print from SMT parser (PR #543): Thanks to **Ahmed Irfan** for providing this patch.

Solvers:

- Yices: Upgrade to 2.6.0 (PR #509).

- Boolector: Upgrade to 3.0.1-pre (7f5d32) (PR #514)

- CVC4: Upgrade to 1.7-prerelease (PR #552) *Known issue*: Passing options to CVC4 fails sometimes.

- Z3: Upgrade to 4.8.4 (PR #550). Removed support for interpolation. *Known issue*: Some tests on use of tactics exhibit some random

> failures on Travis.

- Yices: Add support for OSX (PR #486). Thanks to **Varun Patro** for contributing the patch.

- SMTLIB Solver (PR #524): Add support for custom sorts in SMT-LIB interface. Thanks to **Yoni Zohar** for contributing the patch.

- MathSAT (PR #526): Add option to support preferred variables with polarity. Thanks to **Cristian Mattarei** for providing the patch.

Bugfix:

- SmtLib parser (PR #521): Fix StopIteration error. The error would make it impossible to use the parser with Python 3.7. The fix changes the structure of the parser, in order to separate cases in which we know that there is a token to consume (function consume) and when we want to consume a token only if available (function consume_maybe). Thanks to **@samuelkolb** and **Kangjing Huang** for reporting this.

- Boolector: Fixed bug in LShl and LShr conversion (PR #534)

- Z3 (PR #530, #528): Fixed race condition during context deletion. The race condition would cause pySMT to segfault on certain situations. Thanks to **Haozhong Zhang** for helping us reproduce the issue and to **@Johan-vdberg** for reporting it.

- MathSAT (PR #518): Fix installation error on darwin. Thanks to **Lenny Truong** for contributing the patch.

- Fix declare-sort bug (PR #501). Thanks to **Yoni Zohar** for contributing the patch.

- Fix docstring for BVAShr (PR #503). Thanks to **Mathias Preiner** for contributing the patch.

- Fix yices compilation on OSX without AVX2 instruction (PR #491)

- Fix PysmtTypeError when reusing symbols in SMT-LIB define-fun (PR #502). Thanks to **Yoni Zohar** for contributing the patch.

- Fix doublequote escaping (PR #489). Thanks to **Lukas Dresel** for contributing the patch.

- Fix pySMT CLI for Python3 (PR #493). Thanks to **Radomir Stevanovic** for contributing the patch.

2.3.4 0.7.5: 2018-05-29 – Strings and Cython Parser

General: * Strings Theory (#458)

Add support for the theory of Strings as supported by CVC4.

Direct solver support is limited to CVC4, but the SMT-LIB interface can be used to integrate with other solvers (e.g., Z3).

This feature was largely implemented by **Suresh Goduguluru** and motivated by **Clark Barrett**.

- SMT-LIB Parser: Improved performance with Cython (PR #432)

The SMT-LIB parser module is now compiled using Cython behind the scenes. By default pySMT will try to use the cython version but the behavior can be controlled via env variables:

```
PYSMT_CYTHON=False # disable Cython
PYSMT_CYTHON=True  # force Cython: Raises an error if cython or the
                   # SMT-LIB parser module are not available.
unset PYSMT_CYTHON # defaults to Cython but silently falls back to
                   #pure-python version
```

The API of `pysmt.smtlib.parser` does not change and preserves compatibility with previous versions.

Benchmarking on parse_all.py shows:

```
$ PYSMT_CYTHON=True python3.5 parse_all.py --count 500
The mean execution time was 2.34 seconds
The max execution time was 59.77 seconds

$ PYSMT_CYTHON=False python3.5 parse_all.py --count 500
The mean execution time was 3.39 seconds
The max execution time was 85.46 seconds
```

- SMT-LIB Parser: Added Debugging Information (Line/Col number) (PR #430)

- pysmt-install: Simplified solver version check (PR #431)

- Extended infix notation to support: - Store and Select (PR #437) - NotEquals (PR #438) - EUF Function application (PR #445)

- Examples: Quantifier Elimination in LRA (PR #447)

- Sorts: Stronger type checking for composite sorts (PR #449)

- BvToNatural: Introduced new operator to convert bitvectors into natural numbers (PR #450)

- Examples: Theory Combination (PR #451)

- QE: Introduce new QE techniques based on Self-Substitution (PR #460)

Solvers: * Z3: Upgrade to 4.5.1 dev (082936bca6fb) (PR #407)

- CVC4: Upgrade to 1.5 (PR #424)

- MathSAT: Upgrade to 5.5.1 (PR #453)

- MathSAT: Add Windows Support (PR #453)

Theories: * Support for Theory of Strings (SMT-LIB + CVC4) (PR #458)

Bugfix:

- Z3: Conversion of top-level ITE (PR #433)

- Z3: Fixed exception handling (PR #473): Thanks to **Bradley Ellert** for reporting this.

- Detect BV type in Array and Function when using infix notation (PR #436)

- Support GMPY objects in BV construction (PR #441)

- SMT-LIB: Fixed parsing of #x BV constants (PR #443): Thanks to **@cdmcdonell** for reporting this.

- SMT-LIB: Remove trailing whitespace from bvrol and bvsext (PR #459)

- Fixed type-checking of Equals, LT and LE (PR #452)

- Examples: Revised Einstein example (PR #448): Thanks to **Saul Fuhrmann** for reporting the issue.

- Examples: Fixed indexing and simple path condition in MC example (PR 454): Thanks to **Cristian Mattarei** for contributing this patch.

- Fixed installer for picosat to use HTTPS (PR #481)

2.3.5 0.7.0: 2017-08-12 – Class Based Walkers and Sorts

BACKWARDS INCOMPATIBLE CHANGES:

- Removed option "quantified" in Solver (PR #377)

- Removed deprecated CNFizer.printer method (PR #359)

General:

- Class-Based Walkers (PR #359):

  Walkers behavior is now defined in the class definition. Processing an AND node always calls walk_and. This makes it possible to subclass and override methods, but at the same time call the implementation of a super class, e.g.:

  ```python
  def walk_and(...):
      return ParentWalker.walk_and(self, ....)
  ```

  The utility method Walker.super is provided to automatically handle the dispatch of a node to the correct walk_* function, e.g.,:

  ```python
  def walk_bool_to_bool(...):
      return ParentWalker._super(self, ....)
  ```

  The method Walker.set_functions is deprecated, but kept for compatibility with old-style walkers. Using set_functions has the same effect as before. However, you cannot modify a subclass of a walker using set_functions. *You should not be using set_functions anymore!*

  The method Walker.set_handler is used to perform the same operation of set_function at the class level. The associated decorator @handles can be used to associate methods with nodetypes.

  These changes make it possible to extend the walkers from outside pySMT, without relying on hacks like the Dynamic Walker Factory (DWF). See examples/ltl.py for a detailed example.

- Introduce the support for custom sorts (PySMTTypes) (PR #375)

  Two new classes are introduced: _Type and PartialType

  PartialType is used to represent the concept of SMT-LIB "define-sort". The class _TypeDecl is used to represents a Type declaration, and as such cannot be used directly to instantiate a Symbol. This capture the semantics of declare-sort. A wrapper Type() is given to simplify its use, and making 0-arity sorts a special case. The following two statements are equivalent:

  ```python
  Type("Colors")
  Type("Colors", 0)
  ```

  0-ary type are instantiated by default. For n-ary types, the type needs to be instantiated. This can be done with the method `TypeManager.get_type_instance` or by using infix notation (if enabled):

  ```python
  type_manager.get_type_instance(Type(Pair, 2), Int, Int))
  Type(Pair, 2)(Int, Int)
  ```

  Type declarations and Type instances are memoized in the environment, and suitable shortcuts have been introduced. Logics definition has been extended with the field `custom_types` to detect the use of custom types. *Note*: Due to the limited support of custom types by solvers, by default every SMT-LIB logic is defined with `custom_types=False`.

- Add shortcuts.to_smtlib() to easily dump an SMT-LIB formula

- Add explicit support for BV and UFBV logics (PR #423): Thanks to **Alexey Ignatiev** for reporting this.

Solvers:

- PicoSAT: Upgrade to 965 (PR #425)

- Boolector: Upgrade to 2.4.1 (PR #422)

- CVC4: Fixed memory-leak (PR #419)

- Yices: Upgrade to 2.5.2 (PR #426)

Bugfix:

- Fixed assumption handling in the Boolector wrapper. Thanks to **Alexey Ignatiev** for contributing with this patch!

- Fix cyclic imports (PR #406). Thanks to **@rene-rex** for reporting this.

- Fixed SMT-LIB Script serialization to default to a daggified representation. (PR #418)

- Fixed SMT-LIB Parsing of declare-const . Thanks to **@ahmedirfan1983** for reporting this. (PR #429)

- Fixed logic detection when calling is_unsat (PR #428)

2.3.6 0.6.1: 2016-12-02 – Portfolio and Coverage

General:

- Portfolio Solver (PR #284):

  Created Portfolio class that uses multiprocessing to solve the problem using multiple solvers. get_value and get_model work after a SAT query. Other artifacts (unsat-core, interpolants) are not supported. Factory.is_* methods have been extended to include *portfolio* key-word, and exported as is_* shortcuts. The syntax becomes:

  ```
  is_sat(f, portfolio=["s1", "s2"])
  ```

- Coverage has been significantly improved, thus giving raise to some clean-up of the tests and minor bug fixes. Thanks to Coveralls.io for providing free coverage analysis. (PR #353, PR #358, PR #372)

- Introduce PysmtException, from which all exceptions must inherit. This also introduces hybrid exceptions that inherit both from the Standard Library and from PysmtException (i.e., PysmtValueError). Thanks to **Alberto Griggio** for suggesting this change. (PR #365)

- Windows: Add support for installing Z3. Thanks to **Samuele Gallerani** for contributing this patch. (PR #385)

- Arrays: Improved efficiency of array_value_get (PR #357)

- Documentation: Thanks to the **Hacktoberfest** for sponsoring these activities:

  - Every function in shortcuts.py now has a docstring! Thanks to **Vijay Raghavan** for contributing this patch. (PR #363)

  - Contributing information has been moved to the official documentation and prettyfied! Thanks to **Jason Taylor Hodge** for contributing this patch. (PR #339)

  - Add link to Google Group in Readme.md . Thanks to @ankit01ojha for contributing this. (PR #345)

- smtlibscript_from_formula(): Allow the user to specify a custom logic. Thanks to **Alberto Griggio** for contributing this patch. (PR #360)

Solvers:

- MathSAT: Improve back-conversion performance by using MSAT_TAGS (PR #379)

- MathSAT: Add LIA support for Quantifier Elimination

- Removed: Solver.declare_variable and Solver.set_options (PR #369, PR #378)

Bugfix:

- CVC4:

  - Enforce BV Division by 0 to return a known value (0xFF) (PR #351)

– Force absolute import of CVC4. Thanks to **Alexey Ignatiev** (@2sev) for reporting this issue. (PR #382)

- MathSAT: Thanks to **Alberto Griggio** for contributing these patches

    – Fix assertions about arity of BV sign/zero extend ops. (PR #350, PR #351)

    – Report the error message generated by MathSAT when raising a SolverReturnedUnknownResultError (PR #355)

- Enforce a single call to is_sat in non-incremental mode (PR #368). Thanks to @colinmorris for pointing out this issue.

- Clarified Installation section and added example of call to `pysmt-install --env`. Thanks to **Marco Roveri** (@marcoroveri) for pointing this out.

- SMT-LIB Parser:

    – Minor fixes highlighted by fuzzer (PR #376)

    – Fixed annotations parsing according to SMTLib rules (PR #374)

- pysmt-install: Gracefully fail if GIT is not installed (PR #390) Thanks to **Alberto Griggio** for reporting this.

- Removed dependency from internet connections when checking picosat version (PR #386)

2.3.7 0.6.0: 2016-10-09 – GMPY2 and Goodbye Recursion

BACKWARDS INCOMPATIBLE CHANGES:

- Integer, Fraction and Numerals are now defined in pysmt.constants (see below for details). The breaking changes are:

    – Users should use pysmt.constants.Fraction, if they want to guarantee that the same type is being used (different types are automatically converted);

    – Methods from pysmt.utils moved to pysmt.constants;

    – Numerals class was moved from pysmt.numeral (that does not exist anymore).

- Non-Recursive TreeWalker (PR #322)

  Modified TreeWalker to be non-recursive. The algorithm works by keeping an explicit stack of the walking functions **that are now required to be generators**. See pysmt.printer.HRPrinter for an example. This removes the last piece of recursion in pySMT !

- Times is now an n-ary operator (Issue #297 / PR #304)

  Functions operating on the args of Times (e.g., rewritings) should be adjusted accordingly.

- Simplified module pysmt.parsing into a unique file (PR #301)

  The pysmt.parsing module was originally divided in two files: pratt.py and parser.py. These files were removed and the parser combined into a unique parsing.py file. Code importing those modules directly needs to be updated.

- Use solver_options to specify solver-dependent options (PR #338):

    – MathSAT5Solver option 'debugFile' has been removed. Use the solver option: "debug_api_call_trace_filename".

    – BddSolver used to have the options as keyword arguments (static_ordering, dynamic_reordering etc). This is not supported anymore.

- Removed deprecated methods (PR #332):

    – FNode.get_dependencies (use FNode.get_free_variables)

- FNode.get_sons (use FNode.get_args)

- FNode.is_boolean_operator (use FNode.is_bool_op)

- pysmt.test.skipIfNoSolverAvailable

- pysmt.randomizer (not used and broken)

General:

- Support for GMPY2 to represent Fractions (PR #309).

  Usage of GMPY2 can be controlled by setting the env variable PYSMT_GMPY to True or False. By default, pySMT tries to use GMPY2 if installed, and fallbacks on Python's Fraction otherwise.

- Constants module: pysmt.constants (PR #309)

  This module provides an abstraction for constants Integer and Fraction, supporting different ways of representing them internally. Additionally, this module provides several utility methods:

  - is_pysmt_fraction

  - is_pysmt_integer

  - is_python_integer

  - is_python_rational

  - is_python_boolean

  Conversion can be achieved via:

  - pysmt_fraction_from_rational

  - pysmt_integer_from_integer

  - to_python_integer (handle long/int py2/py3 mismatch)

- Add Version information (Issue #299 / PR #303)

  - pysmt.VERSION : A tuple containing the version information

  - pysmt.__version__ : String representation of VERSION (following PEP 440)

  - pysmt.git_version : A simple function that returns the version including git information.

  install.py (pysmt-install) and shell.py gain a new –version option that uses git_version to display the version information.

- Shortcuts: read_smtlib() and write_smtlib()

- Docs: Completely Revised the documentation (PR #294)

- Rewritings: TimesDistributor (PR #302)

  Perform distributivity on an N-ary Times across addition and subtraction.

- SizeOracle: Add MEASURE_BOOL_DAG measure (PR #319)

  Measure the Boolean size of the formula. This is equivalent to replacing every theory expression with a fresh boolean variable, and measuring the DAG size of the formula. This can be used to estimate the Boolean complexity of the SMT formula.

- PYSMT_SOLVERS controls available solvers (Issue #266 / PR #316):

  Using the PYSMT_SOLVER system environment option, it is possible to restrict the set of installed solvers that are actually accessible to pySMT. For example, setting PYSMT_SOLVER="msat,z3" will limit the accessible solvers to msat and z3.

- Protect FNodeContent.payload access (Issue #291 / PR 310)

  All methods in FNode that access the payload now check that the FNode instance is of the correct type, e.g.:

  FNode.symbol_name() checks that FNode.is_symbol()

  This prevents from accessing the payload in a spurious way. Since this has an impact on every access to the payload, it has been implemented as an assertion, and can be disabled by running the interpreter with -O.

Solvers:

- Z3 Converter Improvements (PR #321):

  - Optimized Conversion to Z3 Solver Forward conversion is 4x faster, and 20% more memory efficient, because we work at a lower level of the Z3 Python API and do not create intermediate AstRef objects anymore. Back conversion is 2x faster because we use a direct dispatching method based on the Z3 OP type, instead of the big conditional that we were using previously.

  - Add back-conversion via SMT-LIB string buffer. Z3Converter.back_via_smtlib() performs back conversion by printing the formula as an SMT-LIB string, and parsing it back. For formulas of significant size, this can be drastically faster than using the API.

  - Extend back conversion to create new Symbols, if needed. This always raise a warning alerting the user that a new symbol is being implicitly defined.

- OSX: Z3 and MathSAT can be installed with pysmt-install (PR #244)

- MathSAT: Upgrade to 5.3.13 (PR #305)

- Yices: Upgrade to 2.5.1

- Better handling of solver options (PR #338):

  Solver constructor takes the optional dictionary `solver_options` of options that are solver dependent. It is thus possible to directly pass options to the underlying solver.

Bugfix:

- Fixed: Times back conversion in Z3 was binary not n-ary. Thanks to **Ahmed Irfan** for submitting the patch (PR #340, PR #341)

- Fixed: Bug in `array_value_assigned_values_map`, returning the incorrect values for an Array constant value. Thanks to **Daniel Ricardo dos Santos** for pointing this out and submitting the patch.

- Fixed: SMT-LIB define-fun serialization (PR #315)

- Issue #323: Parsing of variables named bvX (PR #326)

- Issue #292: Installers: Make dependency from pip optional (PR #300)

- Fixed: Bug in MathSAT's `get_unsat_core` (PR #331), that could lead to an unbounded mutual recursion. Thanks to **Ahmed Irfan** for reporting this (PR #331)

2.3.8 0.5.1: 2016-08-17 – NIRA and Python 3.5

Theories:

- Non Linear Arithmetic (NRA/NIA): Added support for non-linear, polynomial arithmetic. This thoery is currently supported only by Z3. (PR #282)

  - New operator POW and DIV

  - LIRA Solvers not supporting Non-Linear will raise the NonLinearError exception, while solvers not supporting arithmetics will raise a ConvertExpressionError exception (see test_nlira.py:test_unknownresult)

> – Algebraic solutions (e.g., sqrt(2) are represented using the internal z3 object – This is bound to change in the future.

General:

- Python 3.5: Full support for Python 3.5, all solvers are now tested (and working) on Python 3.5 (PR #287)
- Improved installed solvers check (install.py)

  - install.py –check now takes into account the bindings_dir and prints the version of the installed solver
  - Bindings are installed in different directories depending on the minor version of Python. In this way it is possible to use both Python 2.7 and 3.5.
  - There is a distinction btw installed solvers and solvers in the PYTHONPATH.
  - Qelim, Unsat-Core and Interpolants are also visualized (but not checked)

- Support for reading compressed SMT-LIB files (.bz2)
- Simplified HRPrinter code
- Removed six dependency from type_checker (PR #283)
- BddSimplifier (pysmt.simplifier.BddSimplifier): Uses BDDs to simplify the boolean structure of an SMT formula. (See test_simplify.py:test_bdd_simplify) (PR #286)

Solvers:

- Yices: New wrapper supporting python 3.5 (https://github.com/pysmt/yicespy)
- Yices: Upgrade to 2.4.2
- SMT-LIB Wrapper: Improved interaction with subprocess (#298)

Bugfix:

- Bugfix in Z3Converter.walk_array_value. Thanks to **Alberto Griggio** for contributing this patch
- Bugfix in DL Logic comparison (commit 9e9c8c)

2.3.9 0.5.0: 2016-06-09 – Arrays

BACKWARDS INCOMPATIBLE CHANGES:

- MGSubstituter becomes the new default substitution method (PR #253)

  When performing substitution with a mapping like `{a:  b, Not(a), c}`, `Not(a)` is considered before `a`. The previous behavior (MSSubstituter) would have substituted `a` first, and then the rule for `Not(a)` would not have been applied.

- Removed argument `user_options` from Solver()

Theories:

- Added support for the Theory of Arrays.

  In addition to the SMT-LIB definition, we introduce the concept of Constant Array as supported by MathSAT and Z3. The theory is currently implemented for MathSAT, Z3, Boolector, CVC4.

  Thanks to **Alberto Griggio**, **Satya Uppalapati** and **Ahmed Irfan** for contributing through code and discussion to this feature.

General:

- Simplifier: Enable simplification if IFF with constant: e.g., (a <-> False) into !a

---

- Automatically enable Infix Notation by importing shortcuts.py (PR #267)

- SMT-LIB: support for define-sort commands without arguments

- Improved default options for shortcuts:

  - Factory.is_* sets model generation and incrementality to False;

  - Factory.get_model() sets model generation to True, and incrementality to False.

  - Factory.Solver() sets model generation and incrementality to True;

- Improved handling of options in Solvers (PR #250):

  Solver() takes `**options` as free keyword arguments. These options are checked by the class SolverOptions, in order to validate that these are meaningful options and perform a preliminary validation to catch typos etc. by raising a ValueError exception if the option is unknown.

  It is now possible to do: `Solver(name="bdd", dynamic_reordering=True)`

Solvers:

- rePyCUDD: Upgrade to 75fe055 (PR #262)

- CVC4: Upgrade to c15ff4 (PR #251)

- CVC4: Enabled Quantified logic (PR #252)

Bugfixes:

- Fixed bug in Non-linear theories comparison

- Fixed bug in reset behavior of CVC4

- Fixed bug in BTOR handling of bitwidth in shifts

- Fixed bug in BTOR's get_value function

- Fixed bug in BTOR, when operands did not have the same width after rewriting.

2.3.10 0.4.4: 2016-05-07 – Minor

General:

- BitVectors: Added support for infix notation

- Basic performance optimizations

Solvers:

- Boolector: Upgraded to version 2.2.0

Bugfix:

- Fixed bug in ExactlyOne args unpacking. Thanks to **Martin** @hastyboomalert for reporting this.

2.3.11 0.4.3: 2015-12-28 – Installers and HR Parsing

General:

- pysmt.parsing: Added parser for Human Readable expressions

- pysmt-install: new installer engine

- Most General Substitution: Introduced new Substituter, that performs top-down substitution. This will become the default in version 0.5.

- Improved compliance with SMT-LIB 2 and 2.5

- EagerModel can now take a solver model in input

- Introduce new exception 'UndefinedSymbolError' when trying to access a symbol that is not defined.

- Logic names can now be passed to shortcuts methods (e.g., is_sat) as a string

Solvers:

- MathSAT: Upgraded to version 5.3.9, including support for new detachable model feature. Thanks to **Alberto Griggio** for contributing this code.

- Yices: Upgraded to version 2.4.1

- Shannon: Quantifier Elimination based on shannon expansion (shannon).

- Improved handling of Context ('with' statement), exit and __del__ in Solvers.

Testing:

- Introduced decorator pysmt.test.skipIfNoSMTWrapper

- Tests do note explicitly depend anymore on unittest module. All tests that need to be executable only need to import pysmt.test.main.

Bugfix:

- #184: MathSAT: Handle UF with boolean args Fixed incorrect handling of UF with bool arguments when using MathSAT. The converter now takes care of rewriting the formula.

- #188: Auto-conversion of 0-ary functions to symbols

- #204: Improved quoting in SMT-LIB output

- Yices: Fixed a bug in push() method

- Fixed bug in Logic name dumping for SMT-LIB

- Fixed bug in Simplifier.walk_plus

- Fixed bug in CNF Converter (Thanks to Sergio Mover for pointing this out)

Examples:

- parallel.py: Shows how to use multi-processing to perform parallel and asynchronous solving

- smtlib.py: Demonstrates how to perform SMT-LIB parsing, dumping and extension

- einstein.py: Einstein Puzzle with example of debugging using UNSAT-Cores.

2.3.12 0.4.2: 2015-10-12 – Boolector

Solvers:

- Boolector 2.1.1 is now supported

- MathSAT: Updated to 5.3.8

General:

- EqualsOrIff: Introduced shortcut to handle equality and mismatch between theory and predicates atoms. This simply chooses what to use depending on the operands: Equals if Theory, Iff if predicates. Example usage in examples/all_smt.py

- Environment Extensibility: The global classes defined in the Environment can now be replaced. This makes it much easier for external tools to define new FNode types, and override default services.

- Parser Extensibility: Simplified extensibility of the parser by splitting the special-purpose code in the main loop in separate functions. This also adds support for escaping symbols when dealing with SMT-LIB.

- AUTO Logic: Factory methods default to logics.AUTO, providing a smarter selection of the logic depending on the formula being solved. This impacts all is_* functions, get_model, and qelim.

- Shell: Import BV32 and BVType by default, and enable infix notation

- Simplified HRPrinter

- Added AIG rewriting (rewritings.AIGer)

Bugfix:

- Fixed behavior of CNFizer.cnf_as_set()

- Fixed issue #159: error in parsing let bindings that refer to previous let-bound symbols. Thanks to *Alberto Griggio* for reporting it!

2.3.13 0.4.1: 2015-07-13 – BitVectors Extension

Theories:

- BitVectors: Added Signed operators

Solvers:

- Support for BitVectors added for Z3, CVC4, and Yices

General:

- SmartPrinting: Print expression by replacing sub-expression with custom strings.

- Moved global environment initialization to environment.py. Now internal functions do no need to import short-cuts.py anymore, thus breaking some circular dependencies.

Deprecation:

- Started deprecation of get_dependencies and get_sons

- Depreaced Randomizer and associated functions.

2.3.14 0.4.0: 2015-06-15 – Interpolation and BDDs

General:

- Craig interpolation support through Interpolator class, binary_interpolant and sequence_interpolant shortcuts. Current support is limited to MathSAT and Z3. Thanks to Alberto Griggio for implementing this!

- Rewriting functions: nnf-ization, prenex-normalization and disjunctive/conjunctive partitioning.

- get_implicant(): Returns the implicant of a satisfiable formula.

- Improved support for infix notation.

- Z3Model Iteration bugfix

BDDs:

- Switched from pycudd wrapper to a custom re-entrant version called repycudd (https://github.com/pysmt/repycudd)

- Added BDD-Based quantifier eliminator for BOOL theory

- Added support for static/dynamic variable ordering

- Re-implemented back-conversion avoiding recursion

2.3.15 0.3.0: 2015-05-01 – BitVectors/UnsatCores

Theories:

- Added initial support for BitVectors and QF_BV logic. Current support is limited to MathSAT and unsigned operators.

Solvers:

- Two new quantifier eliminators for LRA using MathSAT API: Fourier-Motzkin (msat_fm) and Loos-Weisspfenning (msat_lw)
- Yices: Improved handling of int/real precision

General:

- Unsat Cores: Unsat core extraction with dedicated shortcut get_unsat_core . Current support is limited to MathSAT and Z3
- Added support for Python 3. The library now works with both Python 2 and Python 3.
- QuantifierEliminator and qelim shortcuts, as well as the respective factory methods can now accept a 'logic' parameter that allows to select a quantifier eliminator instance supporting a given logic (analogously to what happens for solvers).
- Partial Model Support: Return a partial model whenever possible. Current support is limited to MathSAT and Z3.
- FNode.size(): Added method to compute the size of an expression using multiple metrics.

2.3.16 0.2.4: 2015-03-15 – PicoSAT

Solvers:

- PicoSAT solver support

General:

- Iterative implementation of FNode.get_free_variables(). This also deprecates FNode.get_dependencies().

Bugfix:

- Fixed bug (#48) in pypi package, making pysmt-install (and other commands) unavailable. Thanks to Rhishikesh Limaye for reporting this.

2.3.17 0.2.3: 2015-03-12 – Logics Refactoring

General:

- install.py: script to automate the installation of supported solvers.
- get_logic() Oracle: Detects the logic used in a formula. This can now be used in the shortcuts (_is_sat()_, _is_unsat()_, _is_valid()_, and _get_model()_) by choosing the special logic pysmt.logics.AUTO.
- Expressions: Added Min/Max operators.
- SMT-LIB: Substantially improved parser performances. Added explicit Annotations object to deal with SMT-LIB Annotations.

- Improved iteration methods on EagerModel

Backwards Incompatible Changes:

- The default logic for Factory.get_solver() is now the most generic *quantifier free* logic supported by pySMT (currently, QF_UFLIRA). The factory not provides a way to change this default.

- Removed option _quantified_ from all shortcuts.

2.3.18 0.2.2: 2015-02-07 – BDDs

Solvers:

- pyCUDD to perform BDD-based reasoning

General:

- Dynamic Walker Function: Dynamic Handlers for new node types can now be registered through the environment (see Environment.add_dynamic_walker_function).

2.3.19 0.2.1: 2014-11-29 – SMT-LIB

Solvers:

- Yices 2

- Generic Wrapper: enable usage of any SMT-LIB compatible solver.

General:

- SMT-LIB parsing

- Changed internal representation of FNode

- Multiple performance improvements

- Added configuration file

2.3.20 0.2.0: 2014-10-02 – Beta release.

Theories: LIRA Solvers: CVC4 General:

- Type-checking

- Definition of SMT-LIB logics

- Converted the DAGWalker from recursive to iterative

- Better handling of errors during formula creation and solving

- Preferences among available solvers.

Deprecation:

- Option 'quantified' within Solver() and all related methods will be removed in the next release.

Backwards Incompatible Changes:

- Renamed the module pysmt.types into pysmt.typing, to avoid conflicts with the Python Standard Library.

2.3.21 0.1.0: 2014-03-10 – Alpha release.

Theories: LIA, LRA, RDL, EUF Solvers: MathSAT, Z3 General Functionalities:

- Formula Manipulation: Creation, Simplification, Substitution, Printing
- Uniform Solving for QF formulae
- Unified Quantifier Elimination (Z3 support only)

2.3.22 0.0.1: 2014-02-01 – Initial release.

2.4 Developing in pySMT

2.4.1 Licensing

pySMT is distributed under the APACHE License (see LICENSE file). By submitting a contribution, you automatically accept the conditions described in LICENSE. Additionally, we ask you to certify that you have the right to submit such contributions. We adopt the "Developer Certificate of Origin" approach as done by the Linux kernel.

```
Developer Certificate of Origin Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors. 660
York Street, Suite 102, San Francisco, CA 94110 USA

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have
    the right to submit it under the open source license indicated in
    the file; or

(b) The contribution is based upon previous work that, to the best of my
    knowledge, is covered under an appropriate open source license and I
    have the right under that license to submit that work with
    modifications, whether created in whole or in part by me, under the
    same open source license (unless I am permitted to submit under a
    different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person
    who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are
    public and that a record of the contribution (including all personal
    information I submit with it, including my sign-off) is maintained
    indefinitely and may be redistributed consistent with this project
    or the open source license(s) involved.
```

During a Pull-Request you will be asked to edit the CONTRIBUTORS file to add your name and email address. By doing so, you agree to the CLA. You will only have to complete this once, but this applies to **all** your contributions.

2.4.2 Tests

Running Tests

Tests in pySMT are developed using python's built-in testing framework *unittest*. Each *TestCase* is stored into a separate file, and it should be possible to launch it by calling the file directly, e.g.: `$ python test_formula. py`.

However, the preferred way is to use pytest, e.g.: `$ python -m pytest pysmt/tests/test_formula. py`.

There are two utility scripts to simplify the testing of pysmt: `run_tests.sh` and `run_all_tests.sh`. They both exploit additional options for pytest, such as timeouts. `run_all_tests.sh` includes all the tests that are marked as `slow`, and therefore might take some time to complete.

Finally, tests are run across a wide range of solvers, versions of python and operating systems using Travis CI. This happens automatically when you open a PR. If you want to run this before submitting a PR, create a (free) Travis CI account, fork pySMT, and enable the testing from Travis webinterface.

All tests should pass for a PR to be merged.

Writing Tests

TestCase should inherit from `pysmt.test.TestCase`. This provides a default `SetUp()` for running tests in which the global environment is reset after each test is executed. This is necessary to avoid interaction between tests. Moreover, the class provides some additional assertions:

PYSMT_SOLVER

The system environment variable `PYSMT_SOLVER` controls which solvers are actually available to pySMT. When developing it is common to have multiple solvers installed, but wanting to only test on few of them. For this reason `PYSMT_SOLVER` can be set to a list of solvers, e.g., `PYSMT_SOLVER="msat, z3"` will provide access to pySMT only to msat and z3, independently of which other solvers are installed. If the variable is unset or set to `all`, it does not have any effect.

2.4.3 How to add a new Theory within pySMT

In pySMT we are trying to closely follow the SMT-LIB standard. If the theory you want to add is already part of the standard, than many points below should be easy to answer.

1. **Identify the set of operators that need to be added** You need to distinguish between operators that are needed to represent the theory, and operators that are syntactic sugar. For example, in pySMT we have less-than and less-than-equal, as basic operators, and define greater-than and greater-than-equal as syntactic sugar.

2. **Identify which solvers support the theory** For each solver that supports the theory, it is important to identify which sub/super-set of the operators are supported and whether the solver is already integrated in pySMT. The goal of this activity is to identify possible incompatibilities in the way different solvers deal with the theory.

3. **Identify examples in "SMT-LIB" format** This provides a simple way of looking at how the theory is used, and access to cheap tests.

Once these points are clear, please open an issue on github with the answer to these points and a bit of motivation for the theory. In this way we can discuss possible changes and ideas before you start working on the code.

Code for a new Theory

A good example of theory extension is represented by the BitVector theory. In case of doubt, look at how the BitVector case (bv) has been handled.

Adding a Theory to the codebase is done by following these steps:

1. Tests: Add a test file `pysmt/test/test_<theory>.py`, to demonstrate the use for the theory (e.g., `pysmt/test/test_bv.py`).

2. Operators: Add the (basic) operators in `pysmt/operators.py`, create a constant for each operator, and extend the relevant structures.

3. Typing: Extend `pysmt/typing.py` to include the types (sorts) of the new theory.

4. Walker: Extend `pysmt/walkers/generic.py` to include one `walk_` function for each of the basic operators.

5. FNode: Extend `is_*` methods in `pysmt/fnode.py:FNode`. This makes it possible to check the type of an expression, obtaining additional elements (e.g., width of a bitvector constant).

6. Typechecker: Extend `pysmt/type_checker.py:SimpleTypeChecker` to include type-checking rules.

7. FormulaManager: Create constructor for all operators, including syntactic sugar, in `pysmt/formula.py:FormulaManager`.

At this point you are able to build expressions in the new theory. This is a good time to start running your tests.

8. Printers: Extend `pysmt/printers.py:HRPrinter` to be able to print expressions in the new theory (you might need to do this earlier, if you need to debug your tests output).

9. Examples: Extend `pysmt/test/examples.py` with at least one example formula for each new operator defined in FormulaManager. These examples are used in many tests, and will help you identify parts of the system that still need to be extended (e.g., Simplifier).

10. Theories and Logics: Extend `pysmt/logics.py` to include the new Theory and possible logics that derive from this Theory. In particular, define logics for each theory combination that makes sense.

11. SMT-LIB: Extend `pysmt/smtlib/parser.py:SmtLibParser` and `pysmt/smtlib/printers.py` to support the new operators.

12. Shortcuts: All methods that were added in FormulaManager need to be available in `pysmt/shortcuts.py`.

At this point all pySMT tests should pass. This might require extending other walkers to support the new operators.

13. Solver: Extend at least one solver to support the Logic. This is done by extending the associated Converter (e.g., `pysmt/solvers/msat.py:MSatConverter`) and adding at least one logic to its `LOGICS` field. As a bare-minimum, this will require a way of converting solvers-constants back into pySMT constants (`Converter.back()`).

2.4.4 Packaging and Distributing PySMT

The setup.py script can be used to create packages. The command

```
python setup.py bdist --format=gztar
```

will produce a tar.gz file inside the `dist/` directory.

For convenience the script `make_distrib.sh` is provided, this builds both the binary and source distributions within `dist/`.

2.4.5 Building Documentation

pySMT uses Sphinx for documentation. To build the documentation you will need Sphinx installed, this can be done via pip.

A Makefile in the `docs/` directory allows to build the documentation in many formats. Among them, we usually consider html and latex.

2.4.6 Preparing a Release (Check-List)

In order to make a release, the master branch must pass all tests on the CI (Travis and Appveyor). The release process is broken down into the following steps:

- Release branch creation
- Changelog update
- Version change
- Package creation and local testing
- Merge and Tag
- PyPi update
- Version Bumping
- Announcement

Release Branch Creation

As all other activities, the creation of a release requires working on a separate branch. This makes it possible to interrupt, share, and resume the release creation, if bugs are discovered during this process. The branch must be called `rc/a.b.c`, where a.b.c is the version number of the target release.

Changelog Update (docs/CHANGES.rst)

Use `git log` to obtain the full list of commits from the latest tag. We use merge commits to structure the Changelog, however, sometimes additional and useful information is described in intermediate commits, and it is thus useful to have them.

The format of the header is `<version>: <year> -- <Title>`, where version has the format Major.Minor.Patch (e.g., 0.6.1) and year is in ISO format: YYYY-MM-DD (e.g., 2016-11-28). The title should be brief and possible include the highlights of the release.

The body of the changelog should start with the backwards incompatible changes with a prominent header. The other sections (optional if nothing changed) are:

- General: For new features of pySMT
- Solvers: For upgrades or improvements to the solvers
- Theories: For new or improved Theories
- Bugfix: For all the fixes that do not constitute a new feature

Each item in the lists ends with reference to the Github issue or Pull request. If an item deserves more explanation and it is not associated with an issue or PR, it is acceptable to point to the exact commit id). Items should also acknowledge contributors for submitting patches, opening tickets or simply discussing a problem.

Version change

The variable `VERSION` in `pysmt/__init__.py` must be modified to show the correct version number: e.g., `VERSION = (0, 6, 1)`.

Package creation and local testing

The utility script `make_distrib.sh` to create a distribution package is located in the root directory of the project. This will create various formats.

After running this script, the package `dist/PySMT-a.b.c.tar.gz` (where a.b.c are the release number), needs to be uploaded to pypi. Before doing so, however, we test it locally, to make sure that everything works. The most common mistake in this phase is the omission of a file in the package.

To test the package, we create a new hardcopy of the tests of pySMT:

0. `mkdir -p test_pkg/pysmt`

1. `cp -a github/pysmt/test test_pkg/pysmt/; cd test_pkg`

2. This should fail: `python -m pytest pysmt`

3. `pip install --user github/dist/PySMT-a.b.c.tar.gz`

4. `python -m pytest pysmt`

5. `pip uninstall pysmt`

All tests should pass in order to make the release. Note: It is enough to have one solver installed, in order to test the package. The type of issues that might occur during package creation are usually independent of the solver.

Merge and Tag

At this point we have created and tested the release, we can merge the `rc/` branch back into master, and tag the release with: `git tag -a va.b.c` (note the `v` before the major version number), and finally push the tag to github `git push origin va.b.c`.

Now on github, it is possible to create the release associated with this tag. The description of the release is the copy-paste of the Changelog. Additionally, we include the wheel file and the tar.gz .

Immediately after tagging, make a commit on master bumping the version. By default we use `(a, b, c+1, "dev", 1)`.

PyPi update

`twine upload PySMT-a.b.c.tar.gz`

TODO: Figure out how to have shared credentials for pypi. Currently, only marcogario has upload privileges.

Announcement

- Mailing list: https://groups.google.com/forum/#!forum/pysmt

- Make sure the Github Release has been created

2.4.7 Performance Tricks

It is our experience that in many cases the performance limitations come from the solvers or from a sub-optimal encoding of the problem, and that pySMT performs well for most use-cases. Nevertheless, sometimes you just want to squeeze a bit more performance from the library, and there are a few things that you might want to try. As always, you should make sure that your code is correct before starting to optimize it.

Disable Assertions

Run the python interpreter with the `-O` option. Many functions in pySMT have assertions to help you discover problems early on. By using the command line option `-O` all assertions are disabled

Avoid Infix Notation and shortcuts

Infix notation and shortcuts assume that you are operating on the global environment. The expression `a & b` needs to:

1. Resolve the implicit operator (i.e., translate `&` into `And`)

2. Access the global environment

3. Access the corresponding formula manager

4. Check if the right-hand-side is already an FNode

5. Call `FormulaManager.And` on the two elements.

Using a shortcut is similar in complexity, although you skip step 1 and 4. Therefore, within loop intensive code, make sure that you obtain a reference to the current formula manager or even better to the actual function call that you want to perform: e.g.,

```
Real = get_env().formula_manager.Real
for x in very_large_set:
    Real(x)
```

This will save dereferencing those objects over-and-over again.

Disabling Type-Checking

If you really want to squeeze that extra bit of performance, you might consider disabling the type-checker. In pySMT all expressions are checked at creation time in order to guarantee that they are well-formed and well-typed. However, this also means that on very big expressions, you will call many times the type-checker (see discussion in #400). Although, all calls to the type-checker are memoized, the cost of doing so can add up. If you are 100% sure that your expressions will be well-typed, then you can use the following code to create a context that disables temporarily the type-checker. WARNING: If you create an expression that is not well-typed while the type-checker is disabled, there is no way to detect it later on.

```
class SuspendTypeChecking(object):
    """Context to disable type-checking during formula creation."""

    def __init__(self, env=None):
        if env is None:
            env = get_env()
        self.env = env
        self.mgr = env.formula_manager
```

(continues on next page)

```python
    def __enter__(self):
        """Entering a Context: Disable type-checking."""
        self.mgr._do_type_check = lambda x : x
        return self.env

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Exiting the Context: Re-enable type-checking."""
        self.mgr._do_type_check = self.mgr._do_type_check_real
```

This can be used as follows:

```python
with SuspendTypeChecking():
    r = And(Real(0), Real(1))
```

PyPy

pySMT is compatible with pypy. Unfortunately, we cannot run most of the solvers due to the way the bindings are created today. However, if are interfacing through the SMT-LIB interface, or are not using a solver, you can run pySMT using pypy. This can drastically improve the performances of code in which most of the time is spent in simple loops. A typical example is parsing, modifying, and dumping an SMT-LIB: this flow can significantly improve by using pypy.

Some work has been done in order to use CFFI to interface more solvers with pypy (see mathsat-cffi repo). If you are interested in this activity, please get in touch.

2.5 API Reference

- *Shortcuts*
- *Solver, Model, QuantifierEliminator, Interpolator, and UnsatCoreSolver*
- *Environment*
- *Exceptions*
- *Factory*
- *FNode*
- *Formula*
- *Logics*
- *Operators*
- *Oracles*
- *Parsing*
- *Printers*
- *Simplifier*
- *SMT-LIB*

- *Substituter*

- *Type-Checker*

- *Typing*

- *Walkers*

2.5.1 Shortcuts

2.5.2 Solver, Model, QuantifierEliminator, Interpolator, and UnsatCoreSolver

class `pysmt.solvers.solver.`**Solver**(*environment*, *logic*, ***options*)
    Represents a generic SMT Solver.

OptionsClass
    alias of `pysmt.solvers.options.SolverOptions`

solve(*assumptions=None*)
    Returns the satisfiability value of the asserted formulas.

    Assumptions is a list of Boolean variables or negations of boolean variables. If assumptions is specified, the satisfiability result is computed assuming that all the specified literals are True.

    A call to solve([a1, ..., an]) is functionally equivalent to:

```
push()
add_assertion(And(a1, ..., an))
res = solve()
pop()
return res
```

    but is in general more efficient.

    Other convenience methods (is_sat, is_unsat, is_valid) are wrappers around this function.

    > **Returns** Whether the assertion stack is satisfiable w.r.t. the given assumptions (if given)

    > **Return type** bool

get_model()
    Returns an instance of Model that survives the solver instance.

    Restrictions: Requires option generate_models to be set to true (default) and can be called only after *solve()* (or one of the derived methods) returned sat or unknown, if no change to the assertion set occurred.

is_sat(*formula*)
    Checks satisfiability of the formula w.r.t. the current state of the solver.

    Previous assertions are taken into account.

    > **Returns** Whether formula is satisfiable

    > **Return type** bool

is_valid(*formula*)
    Checks validity of the formula w.r.t. the current state of the solver.

    Previous assertions are taken into account. See *is_sat()*

    > **Returns** Whether formula is valid

Return type bool

is_unsat (*formula*)
    Checks unsatisfiability of the formula w.r.t. the current state of the solver.

    Previous assertions are taken into account. See *is_sat()*

        Returns Whether formula is unsatisfiable

        Return type bool

get_values (*formulae*)
    Returns the value of the expressions if a model was found.

    Requires option generate_models to be set to true (default) and can be called only after *solve()* (or to one of the derived methods) returned sat or unknown, if no change to the assertion set occurred.

        Returns A dictionary associating to each expr a value

        Return type dict

push (*levels=1*)
    Push the current context of the given number of levels.

pop (*levels=1*)
    Pop the context of the given number of levels.

exit ()
    Exits from the solver and closes associated resources.

reset_assertions ()
    Removes all defined assertions.

add_assertion (*formula*, *named=None*)
    Add assertion to the solver.

print_model (*name_filter=None*)
    Prints the model (if one exists).

    An optional function can be passed, that will be called on each symbol to decide whether to print it.

get_value (*formula*)
    Returns the value of formula in the current model (if one exists).

    This is a simplified version of the SMT-LIB function get_values

get_py_value (*formula*)
    Returns the value of formula as a python type.

    E.g., Bool(True) is translated into True. This simplifies writing code that branches on values in the model.

get_py_values (*formulae*)
    Returns the values of the formulae as python types.

    Returns a dictionary mapping each formula to its python value.

class pysmt.solvers.solver.**Model** (*environment*)
    An abstract Model for a Solver.

    This class provides basic services to operate on a model returned by a solver. This class is used as superclass for more specific Models, that are solver dependent or by the EagerModel class.

---

get_value (*formula*, *model_completion=True*)
> Returns the value of formula in the current model (if one exists).

> If model_completion is True, then variables not appearing in the assignment are given a default value, otherwise an error is generated.

> This is a simplified version of the SMT-LIB funtion get_values .

get_values (*formulae*, *model_completion=True*)
> Evaluates the values of the formulae in the current model.

> Evaluates the values of the formulae in the current model returning a dictionary.

get_py_value (*formula*, *model_completion=True*)
> Returns the value of formula as a python type.

> E.g., Bool(True) is translated into True. This simplifies writing code that branches on values in the model.

get_py_values (*formulae*, *model_completion=True*)
> Returns the values of the formulae as python types.

> Returns the values of the formulae as python types. in the current model returning a dictionary.

satisfies (*formula*, *solver=None*)
> Checks whether the model satisfies the formula.

> The optional solver argument is used to complete partial models.

converter
> Get the Converter associated with the Solver.

class `pysmt.solvers.interpolation.`**Interpolator**

binary_interpolant (*a*, *b*)
> Returns a binary interpolant for the pair (a, b), if And(a, b) is unsatisfaiable, or None if And(a, b) is satisfiable.

sequence_interpolant (*formulas*)
> Returns a sequence interpolant for the conjunction of formulas, or None if the problem is satisfiable.

exit ()
> Destroys the solver and closes associated resources.

class `pysmt.solvers.solver.`**UnsatCoreSolver**
> A solver supporting unsat core extraction

get_unsat_core ()
> Returns the unsat core as a set of formulae.

> After a call to solve() yielding UNSAT, returns the unsat core as a set of formulae

get_named_unsat_core ()
> Returns the unsat core as a dict of names to formulae.

> After a call to solve() yielding UNSAT, returns the unsat core as a dict of names to formulae

2.5.3 Environment

2.5.4 Exceptions

This module contains all custom exceptions of pySMT.

exception pysmt.exceptions.**PysmtException**
    Base class for all custom exceptions of pySMT

exception pysmt.exceptions.**UnknownSmtLibCommandError**
    Raised when the parser finds an unknown command.

exception pysmt.exceptions.**SolverReturnedUnknownResultError**
    This exception is raised if a solver returns 'unknown' as a result

exception pysmt.exceptions.**UnknownSolverAnswerError**
    Raised when the a solver returns an invalid response.

exception pysmt.exceptions.**NoSolverAvailableError**
    No solver is available for the selected Logic.

exception pysmt.exceptions.**NonLinearError**
    The provided expression is not linear.

exception pysmt.exceptions.**UndefinedLogicError**
    This exception is raised if an undefined Logic is attempted to be used.

exception pysmt.exceptions.**InternalSolverError**
    Generic exception to capture errors provided by a solver.

exception pysmt.exceptions.**NoLogicAvailableError**
    Generic exception to capture errors caused by missing support for logics.

exception pysmt.exceptions.**SolverRedefinitionError**
    Exception representing errors caused by multiple defintion of solvers having the same name.

exception pysmt.exceptions.**SolverNotConfiguredForUnsatCoresError**
    Exception raised if a solver not configured for generating unsat cores is required to produce a core.

exception pysmt.exceptions.**SolverStatusError**
    Exception raised if a method requiring a specific solver status is incorrectly called in the wrong status.

exception pysmt.exceptions.**ConvertExpressionError**(*message=None*, *expression=None*)
    Exception raised if the converter cannot convert an expression.

exception pysmt.exceptions.**UnsupportedOperatorError**(*message=None*, *node_type=None*, *expression=None*)
    The expression contains an operator that is not supported.

    The argument node_type contains the unsupported operator id.

exception pysmt.exceptions.**SolverAPINotFound**
    The Python API of the selected solver cannot be found.

exception pysmt.exceptions.**UndefinedSymbolError**(*name*)
    The given Symbol is not in the FormulaManager.

exception pysmt.exceptions.**PysmtModeError**
    The current mode is not supported for this operation

exception pysmt.exceptions.**PysmtImportError**

exception pysmt.exceptions.**PysmtValueError**

exception pysmt.exceptions.**PysmtTypeError**

exception pysmt.exceptions.**PysmtSyntaxError**(*message*, *pos_info=None*)

exception pysmt.exceptions.**PysmtIOError**

2.5.5 Factory

2.5.6 FNode

FNode are the building blocks of formulae.

class pysmt.fnode.**FNodeContent**(*node_type*, *args*, *payload*)

    args
        Alias for field number 1

    node_type
        Alias for field number 0

    payload
        Alias for field number 2

class pysmt.fnode.**FNode**(*content*, *node_id*)
    FNode represent the basic structure for representing a formula.

    FNodes are built using the FormulaManager, and should not be explicitly instantiated, since the FormulaManager takes care of memoization, thus guaranteeing that equivalent are represented by the same object.

    An FNode is a wrapper to the structure FNodeContent. FNodeContent defines the type of the node (see operators.py), its arguments (e.g., for the formula A /B, args=(A,B)) and its payload, content of the node that is not an FNode (e.g., for an integer constant, the payload might be the python value 1).

    The node_id is an integer uniquely identifying the node within the FormulaManager it belongs.

    args()
        Returns the subformulae.

    arg(*idx*)
        Return the given subformula at the given position.

    get_free_variables()
        Return the set of Symbols that are free in the formula.

    get_atoms()
        Return the set of atoms appearing in the formula.

    simplify()
        Return a simplified version of the formula.

    substitute(*subs*, *interpretations=None*)
        Return a formula in which subformula have been substituted.

        subs is a dictionary mapping terms to be subtituted with their substitution. interpretations is a dictionary mapping function symbols to an FunctionInterpretation objects describing the semantics of the function.

    size(*measure=None*)
        Return the size of the formula according to the given metric.

        See `SizeOracle`

    get_type()
        Return the type of the formula by calling the Type-Checker.

        See `SimpleTypeChecker`

    is_constant(*_type=None*, *value=None*)
        Test whether the formula is a constant.

Optionally, check that the constant is of the given type and value.

is_bool_constant(*value=None*)
Test whether the formula is a Boolean constant.

Optionally, check that the constant has the given value.

is_real_constant(*value=None*)
Test whether the formula is a Real constant.

Optionally, check that the constant has the given value.

is_int_constant(*value=None*)
Test whether the formula is an Integer constant.

Optionally, check that the constant has the given value.

is_bv_constant(*value=None*, *width=None*)
Test whether the formula is a BitVector constant.

Optionally, check that the constant has the given value.

is_string_constant(*value=None*)
Test whether the formula is a String constant.

Optionally, check that the constant has the given value.

is_algebraic_constant()
Test whether the formula is an Algebraic Constant

is_symbol(*type_=None*)
Test whether the formula is a Symbol.

Optionally, check that the symbol has the given type.

is_literal()
Test whether the formula is a literal.

A literal is a positive or negative Boolean symbol.

is_true()
Test whether the formula is the True Boolean constant.

is_false()
Test whether the formula is the False Boolean constant.

is_toreal()
Test whether the node is the ToReal operator.

is_forall()
Test whether the node is the ForAll operator.

is_exists()
Test whether the node is the Exists operator.

is_quantifier()
Test whether the node is a Quantifier.

is_and()
Test whether the node is the And operator.

is_or()
Test whether the node is the Or operator.

is_not()
Test whether the node is the Not operator.

is_plus()
    Test whether the node is the Plus operator.

is_minus()
    Test whether the node is the Minus operator.

is_times()
    Test whether the node is the Times operator.

is_div()
    Test whether the node is the Division operator.

is_implies()
    Test whether the node is the Implies operator.

is_iff()
    Test whether the node is the Iff operator.

is_ite()
    Test whether the node is the Ite operator.

is_equals()
    Test whether the node is the Equals operator.

is_le()
    Test whether the node is the LE (less than equal) relation.

is_lt()
    Test whether the node is the LT (less than) relation.

is_bool_op()
    Test whether the node is a Boolean operator.

is_theory_relation()
    Test whether the node is a theory relation.

is_theory_op()
    Test whether the node is a theory operator.

is_ira_op()
    Test whether the node is an Int or Real Arithmetic operator.

is_lira_op(***kwargs*)
    Test whether the node is a IRA operator.

is_bv_op()
    Test whether the node is a BitVector operator.

is_array_op()
    Test whether the node is an array operator.

is_bv_not()
    Test whether the node is the BVNot operator.

is_bv_and()
    Test whether the node is the BVAnd operator.

is_bv_or()
    Test whether the node is the BVOr operator.

is_bv_xor()
    Test whether the node is the BVXor operator.

is_bv_concat ()
    Test whether the node is the BVConcat operator.

is_bv_extract ()
    Test whether the node is the BVConcat operator.

is_bv_ult ()
    Test whether the node is the BVULT (unsigned less than) relation.

is_bv_ule ()
    Test whether the node is the BVULE (unsigned less than) relation.

is_bv_neg ()
    Test whether the node is the BVNeg operator.

is_bv_add ()
    Test whether the node is the BVAdd operator.

is_bv_mul ()
    Test whether the node is the BVMul operator.

is_bv_udiv ()
    Test whether the node is the BVUDiv operator.

is_bv_urem ()
    Test whether the node is the BVURem operator.

is_bv_lshl ()
    Test whether the node is the BVLShl (logical shift left) operator.

is_bv_lshr ()
    Test whether the node is the BVLShr (logical shift right) operator.

is_bv_rol ()
    Test whether the node is the BVRol (rotate left) operator.

is_bv_ror ()
    Test whether the node is the BVRor (rotate right) operator.

is_bv_zext ()
    Test whether the node is the BVZext (zero extension) operator.

is_bv_sext ()
    Test whether the node is the BVSext (signed extension) operator.

is_bv_sub ()
    Test whether the node is the BVSub (subtraction) operator.

is_bv_slt ()
    Test whether the node is the BVSLT (signed less-than) operator.

is_bv_sle ()
    Test whether the node is the BVSLE (signed less-than-or-equal-to) operator.

is_bv_comp ()
    Test whether the node is the BVComp (comparison) operator.

is_bv_sdiv ()
    Test whether the node is the BVSDiv (signed division) operator.

is_bv_srem ()
    Test whether the node is the BVSRem (signed reminder) operator.

is_bv_ashr()
    Test whether the node is the BVAshr (arithmetic shift right) operator.

is_select()
    Test whether the node is the SELECT (array select) operator.

is_store()
    Test whether the node is the STORE (array store) operator.

is_array_value()
    Test whether the node is an array value operator.

bv_width()
    Return the BV width of the formula.

bv_extract_start()
    Return the starting index for BVExtract.

bv_extract_end()
    Return the ending index for BVExtract.

bv_rotation_step()
    Return the rotation step for BVRor and BVRol.

bv_extend_step()
    Return the extension step for BVZext and BVSext.

serialize(*threshold=None*)
    Returns a human readable representation of the formula.

    The threshold parameter can be used to limit the amount of the formula that will be printed. See `HRSerializer`

to_smtlib(*daggify=True*)
    Returns a Smt-Lib string representation of the formula.

    The daggify parameter can be used to switch from a linear-size representation that uses 'let' operators to represent the formula as a dag or a simpler (but possibly exponential) representation that expands the formula as a tree.

    See `SmtPrinter`

is_function_application()
    Test whether the node is a Function application.

is_term()
    Test whether the node is a term.

    All nodes are terms, except for function definitions.

symbol_type()
    Return the type of the Symbol.

symbol_name()
    Return the name of the Symbol.

constant_value()
    Return the value of the Constant.

constant_type()
    Return the type of the Constant.

bv2nat()
    Return the unsigned value encoded by the BitVector.

bv_unsigned_value()
> Return the unsigned value encoded by the BitVector.

bv_signed_value()
> Return the signed value encoded by the BitVector.

bv_str(*fmt='b'*)
> Return a string representation of the BitVector.
>
> **fmt: 'b'** [Binary] 'd' : Decimal 'x' : Hexadecimal
>
> The representation is always unsigned

bv_bin_str(*reverse=False*)
> Return the binary representation of the BitVector as string.
>
> The reverse option is provided to deal with MSB/LSB.

array_value_get(*index*)
> Returns the value of this Array Value at the given index. The index must be a constant of the correct type.
>
> This function is equivalent (but possibly faster) than the following code:

```python
m = self.array_value_assigned_values_map()
try:
    return m[index]
except KeyError:
    return self.array_value_default()
```

function_name()
> Return the Function name.

quantifier_vars()
> Return the list of quantified variables.

2.5.7 Formula

2.5.8 Logics

Describe all logics supported by pySMT and other logics defined in the SMTLIB and provides methods to compare and search for particular logics.

class pysmt.logics.**Theory**(*arrays=None, arrays_const=None, bit_vectors=None, floating_point=None, integer_arithmetic=None, real_arithmetic=None, integer_difference=None, real_difference=None, linear=None, uninterpreted=None, custom_type=None, strings=None*)
Describes a theory similarly to the SMTLIB 2.0.

class pysmt.logics.**Logic**(*name, description, quantifier_free=False, theory=None, **theory_kwargs*)
Describes a Logic similarly to the way they are defined in the SMTLIB 2.0

Note: We define more Logics than the ones defined in the SMTLib 2.0. See LOGICS for a list of all the logics and SMTLIB2_LOGICS for the restriction to the ones defined in SMTLIB2.0

get_quantified_version()
> Returns the quantified version of logic.

is_quantified()
> Return whether the logic supports quantifiers.

pysmt.logics.**get_logic_by_name**(*name*)
    Returns the Logic that matches the provided name.

pysmt.logics.**convert_logic_from_string**(*name*)
    Helper function to parse function arguments.

    This takes a logic or a string or None, and returns a logic or None.

pysmt.logics.**get_logic_name**(***logic_kwargs*)
    Returns the name of the Logic that matches the given properties.

    See get_logic for the list of parameters.

pysmt.logics.**get_logic**(*quantifier_free=False*, *arrays=False*, *arrays_const=False*, *bit_vectors=False*, *floating_point=False*, *integer_arithmetic=False*, *real_arithmetic=False*, *integer_difference=False*, *real_difference=False*, *linear=True*, *uninterpreted=False*, *custom_type=False*, *strings=False*)
    Returns the Logic that matches the given properties.

    Equivalent (but better) to executing get_logic_by_name(get_logic_name(...))

pysmt.logics.**most_generic_logic**(*logics*)
    Given a set of logics, return the most generic one.

    If a unique most generic logic does not exists, throw an error.

pysmt.logics.**get_closer_logic**(*supported_logics*, *logic*)
    Returns the smaller supported logic that is greater or equal to the given logic. Raises NoLogicAvailableError if the solver does not support the given logic.

pysmt.logics.**get_closer_pysmt_logic**(*target_logic*)
    Returns the closer logic supported by PYSMT.

pysmt.logics.**get_closer_smtlib_logic**(*target_logic*)
    Returns the closer logic supported by SMT-LIB 2.0.

2.5.9 Operators

This module defines all the operators used internally by pySMT.

Note that other expressions can be built in the FormulaManager, but they will be rewritten (during construction) in order to only use these operators.

pysmt.operators.**new_node_type**(*node_id=None*, *node_str=None*)
    Adds a new node type to the list of custom node types and returns the ID.

pysmt.operators.**op_to_str**(*node_id*)
    Returns a string representation of the given node.

pysmt.operators.**all_types**()
    Returns an iterator over all base and custom types.

2.5.10 Oracles

2.5.11 Parsing

2.5.12 Printers

2.5.13 Simplifier

2.5.14 SMT-LIB

Defines constants for the commands of the SMT-LIB

class pysmt.smtlib.annotations.**Annotations**(*initial_annotations=None*)
Handles and stores (key,value) annotations for formulae

add(*formula*, *annotation*, *value=None*)
Adds an annotation for the given formula, possibly with the specified value

remove(*formula*)
Removes all the annotations for the given formula

remove_annotation(*formula*, *annotation*)
Removes the given annotation for the given formula

remove_value(*formula*, *annotation*, *value*)
Removes the given annotation for the given formula

has_annotation(*formula*, *annotation*, *value=None*)
Returns True iff the given formula has the given annotation. If Value is specified, True is returned only if the value is matching.

annotations(*formula*)
Returns a dictionary containing all the annotations for the given formula as keys and the respective values. None is returned if formula has no annotations.

all_annotated_formulae(*annotation*, *value=None*)
Returns the set of all the formulae having the given annotation key. If Value is specified, only the formula having the specified value are returned.

2.5.15 Substituter

2.5.16 Type-Checker

2.5.17 Typing

This module defines the types of the formulae handled by pySMT.

In the current version these are:

- Bool

- Int

- Real

- BVType

- FunctionType

  • ArrayType

Types are represented by singletons. Basic types (Bool, Int and Real) are constructed here by default, while BVType and FunctionType relies on a factory service. Each BitVector width is represented by a different instance of BVType.

class `pysmt.typing.`**`PySMTType`**(*decl=None*, *basename=None*, *args=None*)
    Class for representing a type within pySMT.

    Instances of this class are used to represent sorts. The subclass FunctionType is used to represent function declarations.

class `pysmt.typing.`**`PartialType`**(*name*, *definition*)
    PartialType allows to delay the definition of a Type.

    A partial type is equivalent to SMT-LIB "define-sort" command.

`pysmt.typing.`**`BVType`**(*width=32*)
    Returns the BV type for the given width.

`pysmt.typing.`**`FunctionType`**(*return_type*, *param_types*)
    Returns Function Type with the given arguments.

`pysmt.typing.`**`ArrayType`**(*index_type*, *elem_type*)
    Returns the Array type with the given arguments.

`pysmt.typing.`**`Type`**(*name*, *arity=0*)
    Returns the Type Declaration with the given name (sort declaration).

2.5.18 Walkers

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index